Reinhold, Paul IF06w1-M

Evaluierung des Eclipse RCP-Frameworks 4.1 auf Produktionstauglichkeit und Einschätzung des Aufwandes für die Umstellung komplexer bestehender Anwendungen

eingereicht als

Masterarbeit

an der

Fachbereich Mathematik/Naturwissenschaften/Informatik

Dresden, 28. Oktober 2011

Erstprüfer:	Prof. Dr. Dirk Labudde
Zweitprüfer:	Dr. Frank Götz

Der Arbeitsprozess wurde begonnen am: 02.05.2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Die Entwicklung von moderner Software wird immer komplexer. Einer der Gründe dafür ist der steigende Umfang der Applikationen. Eine Auswirkung dessen und ein weiterer Grund für die steigenden Komplexität ist die vorrangige Entwicklung der Software innerhalb von Teams. Eine Aufteilung der Software ist unerlässlich, um diesen Prozess effektiv zu gestalten. Unterstützung bei diesem Vorgang bieten Frameworks für die komponentenbasierte Softwareentwicklung in der Form, dass diese eine Aufteilung in unabhängige Teile forcieren und die grundlegende Steuerung der Applikation übernommen wird.

Das auf Plug-ins basierende Eclipse RCP-Framework stellt ein solches dar. Mit dem Release 4.1 beschreitet die neuste Version des Frameworks, den Weg aus dem Early Adopter Status, hin zur nächsten Generation der Entwicklungsplattform für eclipse-basierte Applikationen. Ob das Framework dabei den Ansprüchen der Entwickler gerecht wird, ist Thema dieser Arbeit.

1.1 Analyse des Themas

Um die Aufgabenstellung exakt abzugrenzen, steht die Analyse des Themas an erster Stelle. Für ein gemeinsames Fundament sind zudem die Begrifflichkeiten des Themas zu definieren. Dazu sei das Thema noch einmal genannt:

> Evaluierung des Eclipse RCP-Frameworks 4.1 auf Produktionstauglichkeit und Einschätzung des Aufwandes für die Umstellung komplexer bestehender Anwendungen

Das Thema der Arbeit teil sich in zwei Teilbereiche auf. Der erster Bereich dient der Evaluierung der Produktionstauglichkeit, der zweite einer Einschätzung des Aufwands für eine mögliche Umstellung von Anwendung. Im ersten Teil soll für einen gemeinsamen Standpunkt die Definition der Evaluation der Brockhaus Enzyklopädie [Brock97] gelten.

> *"Analyse und Bewertung eines Sachverhalts, v.a. als Begleitforschung einer Innovation. In diesem Fall ist E. Effizienz- und Erfolgskontrolle zum Zweck der Überprüfung der Eignung eines in Erprobung befindl. Modells."*

Weiterhin weist [Umsta01] in seiner Evaluierungsdefinition darauf hin, dass *"[...] Leistungen, die nicht exakt messbar sind [wie die Produktionstauglichkeit] nach jeweils vorgegebenen Kriterien eingestuft werden"* müssen. Zur Bewertung der Produktionstauglichkeit sind daher geeignete Kriterien zu finden. Diese definiert sich aus der vollständigen Erfüllung der Voraussetzungen unter denen es möglich ist ein Produkt zu entwickeln. Mit der Beantwortung der beiden Fragen sei die Analyse zusammengefasst.

1. Was kann das Eclipse RCP-Framework 4.1 leisten?
2. Was wird für die Entwicklung eines Produktes benötigt?

Die Bewertung der Antworten dieser Fragen schließt Teil 1 des Themas ab. Der sich anschließende Teilbereich beschäftigt sich mit der Umstellung einer Anwendung und der darauf basierenden Einschätzung des Aufwands.

1.2 Zielsetzung der Arbeit

Analog zur Aufteilung des Themas teilen sich die Ziele der Arbeit. Die Auswertung der Fragen *Was kann das Framework?* und *Was wird benötigt?* ist Ziel des ersten Teils, ebenso wie das Finden geeigneter Kriterien zur Bewertung der Produktionstauglichkeit. An dieser Stelle sei angemerkt, dass bei alleiniger Verwendung des Wortes Evaluierung immer der Zusammenhang zur Produktionstauglichkeit zu suchen ist.
Das Ziel des Teils 2 ist mit Hilfe eines praktischen Beispiels die Evaluierung des Frameworks zu unterstützen und eine Grundlage für die Einschätzung des Aufwands zu gewinnen, der nötig ist eine komplexe bestehende Anwendung umzustellen. Das Ergebnis ist dabei nicht Zahlen zu fassen, sondern vielmehr das Treffen einer Aussage über Verhältnismäßigkeit des Aufwands bei der Umstellung gegenüber einer Neuentwicklung.

1.3 Aufbau der Arbeit

Grundlagen zur Entwicklung mir den Eclipse RCP-Framework In diesem einleitenden Kapitel werden die Grundlagen der RCP-Entwicklung vermittelt. Neben den Begriffen von Framework und komponentenbasierter Softwareentwicklung, wird insbesondere auf OSGI, Equinox und Eclipse RCP eingegangen.

Evaluierung des Eclipse RCP-Framework 4.1 Dieses Kapitel stellt den wesentlichen Teil der Arbeit dar. Wie in der Zielsetzung angedeutet, werden die zur Evaluierung dienlichen Fragen beantwortet. Vor allem die Analyse der Neuerungen nehmen einen Großteil in Anspruch. Im letzten Abschnitt des Kapitels findet mit der Extraktion von Entscheidungskriterien eine erste, theorieorientierte Evaluierung des Eclipse RCP-Framework 4.1 statt.

Umstellung komplexer bestehender Anwendungen Thema dieses Kapitels ist die Auswertung der Umstellung einer bestehenden Applikation, sowie eine anschließende Bewertung des Aufwands und eine zweite, praxisorientierte Evaluierung des Eclipse RCP-Frameworks 4.1.

Diskussion und Resultate Inhalt des fünften Kapitels ist die Bewertung der Ergebnisse der Arbeit, konkret der Evaluierung der Produktionstauglichkeit und der Aufwandseinschätzung. Abschluss des Kapitels und der Arbeit ist ein Ausblick auf weitergehende Analysen.

2

Entwicklungsgrundlagen der Eclipse Rich Client Platform

In diesem Kapitel wird auf ein grundlegendes Verständnis des Framworks der Eclipse Rich Client Plattform (RCP) hingeführt. In einer dreistufigen Annäherung wird dabei vom allgemeinen Frameworkbegriff über komponentenbasierende Softwareentwicklung auf die Eigenheiten der Entwicklung mit dem Eclipse RCP-Framework eingegangen. Das Ziel dieses Abschnitts ist die Limitierung des aktuellen Eclipse RCP-Frameworks aufzuzeigen und damit die Notwendigkeit einer Evaluierung der weiterentwickelten Version 4.1 des Frameworks zu begründen. Als Grundlagen zur Entwicklung wird hierbei das allgemeine Verständnis des Entwicklungsrahmens verstanden. Es handelt sich demnach nicht um eine Vorschrift in welcher Art Software mit dem Eclipse RCP-Framework zu entwickeln ist, sondern geht stattdessen auf ausgewählte Merkmale ein.

2.1 Entwicklung mit Frameworks

Als Framework wird eine Programmierstruktur bezeichnet, welche in objektorientierten und komponentenbasierenden Entwicklungsansätzen in der Softwaretechnik häufig Einsatz finden. Wie [Riehle00] treffend formuliert, repräsentiert ein Framework die gesammelte Erfahrung, wie Architektur und Implementierung der meisten Applikationen einer Domäne aussehen sollten. Dabei bleibt jedoch genügend Freiraum für Anpassungen, um das spezielle Problem der Anwendung zu lösen. Frameworks differenzieren sich von Entwurfsmustern dadurch, dass sie deutlich konkreter sind, weil sie bereits in einer bestimmten Sprache implementiert wurden. Entwurfsmuster sind dagegen abstrakt gehalten und können in verschiedenen Sprachen umgesetzt werden.
Der Einsatz eines Frameworks verspricht höhere Produktivität und eine kürzere Time-to-Market. Voraussetzung dafür ist jedoch die entsprechende Kenntnis des Frameworks, das für die Entwicklung genutzt wird. Die bekannten Architektur- und Implemetierungsvorgaben des Frameworks helfen dann den Entwicklungsvorgang zu beschleunigen. Der Aufwand, der für die Einarbeitung in ein Framework nötig ist, zahlt sich i.d.R. durch die verkürzte Entwicklungszeit der Applikation wieder aus. Die Effektivität steigt bei erneuter Verwendung. Frameworks unterscheiden sich von Programm- und Klassen-Bibliotheken durch charakteristische Merkmale [Riehle00].

- *Inversion of Control* – Die Umkehrung der Steuerung. Der Steuerungsfluss wird durch das Framework, nicht durch das Programm, vorgegeben.[1]

- *Standard-Verhalten* Ein Framework ist durch ein nützliches Standard-Verhalten gekennzeichnet, das heißt gleichartige Komponenten und Objekte werden immer in der gleichen Weise erzeugt. Die Einarbeitungszeit verkürzt sich dadurch.

- *Erweiterbarkeit* Ein Framework kann durch den Nutzer um spezielle Funktionen erweitert werden. Dies verspricht sowohl Flexibilität als auch eine langfristig stabile Architektur.

- *Keine Codeveränderung* Der Quellcode eines Frameworks, kann i.d.R. vom Nutzer nicht verändert werden. Der Code ist erweiter-, jedoch nicht veränderbar.

[1] Vgl. Abschnitt Inversion of Control 3.3

Darüber hinaus, existieren verschiedene Typen von Frameworks [ShaHu06]. Die *Application-Frameworks* bieten ein Programmiergerüst für eine bestimmte Klasse von Anwendungen. Sie stellen damit einen horizontalen Schnitt dar, wogegen *Domain-Frameworks* Funktionen und Strukturen eines Problembereichs zur Verfügung stellen und einen vertikalen Schnitt darstellen. Illustrieren lässt sich dies an folgendem Beispiel. Das Eclipse RCP-Framework steht für ein Application-Framework, eines zugeschnitten auf Banksysteme steht dagegen für ein Domain-Framework. Der Übergang dieser Arten ist jedoch fließend, so dass eine strikte Trennung selten möglich ist.
Neben Vorteilen die ein Framework bietet, entstehen bei der Verwendung auch Nachteile. Ein Problem bereitet die Klassenkomplexität. Klassen definieren das Verhalten von Objekten und Instanzen. Diese Objekte stehen für verschiedene Zwecke in zahlreichen Zusammenhängen und reagieren dabei mit aufgabenspezifischem Verhalten. Für derartige Objekte sind auch die Klassen und ihre Verwendung entsprechend komplex. Die hinzukommenden Vererbungshierarchien verschlechtern zudem häufig die Übersichtlichkeit. Um das komplexe Zusammenwirken der Objekte einfacher zu verstehen und bewältigen zu können, ist es naheliegend diese in unabhängige Teile, auch Komponenten genannt, zu zerlegen.

2.2 Komponentenbasierte Softwareentwicklung

Um die steigende Komplexität der Frameworks und der daraus resultierenden Applikationen bewältigen zu können, ist eine Einteilung in so genannte *Softwarekomponenten* sinnvoll. Den Rahmen für die Entwicklung und Ausführung solcher Komponenten bildet ein Komponentenmodell. Dieses Modell legt die Möglichkeiten zur Verknüpfung und Komposition der Komponenten fest, wodurch sich die Komponenten definieren. Zusätzlich bietet das Komponentenmodell in aller Regel eine Infrastruktur an, die häufig benötigte Services zur Verteilung, Persistenz, Nachrichtenaustausch und Sicherheit implementieren [GruTh00]. In der Praxis existieren weitverbreitete Modelle wie Enterprise Java Beans[2], das CORBA Component Model[3] oder OSGi.
Letzteres ist im Rahmen der Arbeit von besonderem Interesse. Es handelt sich um eine offene, modulare und skalierbare *Service Delivery Platform* auf Java-Basis [WüHar08], welche das Fundament der Eclipse Plattform darstellt. Die Abbildung 2.1 zeigt die Architektur des OSGi Modells. Die auf dem Betriebs-

[2] `http://www.oracle.com/technetwork/java/javaee/ejb/index.html`
[3] `http://www.corba.org`

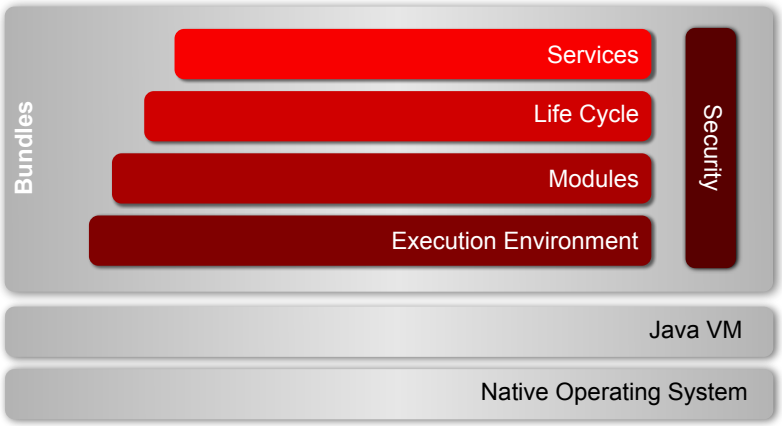

Abbildung 2.1: Architektur des OSGi Modells

system und der Java Virtual Machine (Java VM) aufsetzenden, in Rottönen dargestellten Komponenten werden im OSGi Umfeld als Bundle bezeichnet. Diese Bundles stehen für das Javaumfeld typisch in Form von *.jar*-Archiven bereit. Diese beinhalten neben Klassen und Ressourcen eine Manifestdatei, welche die Abhängigkeit zu anderen Bundles und die öffentliche Schnittstelle des eigenen Bundles festlegt [Seeber08]. Damit ist dieses Mainfest Ausgangspunkt der geforderten Komponenteneigenschaften wie der größtmögliche innere Zusammenhang oder die lose Kopplung zu anderen Komponenten.

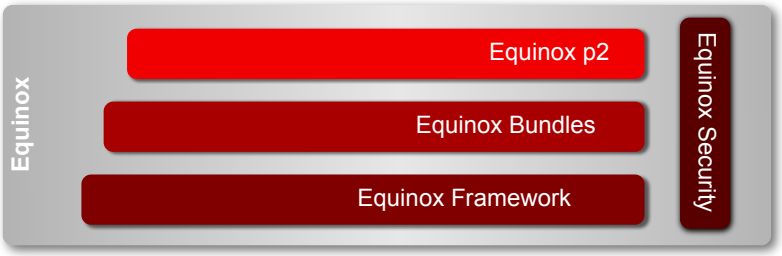

Abbildung 2.2: Komponenten des Equniox-Frameworks

In Form des Equinox-Frameworks übernimmt OSGi in der Entwicklungsumgebung Eclipse die Rolle einer desktop- und enterprise-orientieren Plattform. Das von der Eclipse Foundation[4] entwickelte Framework, wurde 2004 mit Eclipse 3.0 als erste equinox-basierende Version der Entwicklungsumgebung veröffentlicht. Die Abbildung 2.2 stellt einen Auszug der Komponenten des Frameworks dar. Äquivalent zum OGSi Modell basiert das Framework auf dem Betriebssystem und der Java VM.

[4] `http://www.eclipse.org/org/foundation/`

Das als *Equinox-Framework* bezeichnete Bundle beinhaltet die vollständige Umsetzung der OSGi-Spezifikation und stellt weiterhin Launcher, Bootstrap Infrastrukturen[5] und Applikationsmodell bereit.
Die *Equinox Bundles* implementieren alle Add-on Services, welche in der OSGi Spezifikation beschrieben werden. Zusätzlich definieren diese Bundles Services, welche die Programmierung mit dem OSGi System unterstützen. Eine vollständige Aufzeichnung und Erläuterung alle verfügbaren Bundles innerhalb dieser Komponente ist in der Equinox Dokumentation zu finden.[6]
Der als *p2* bezeichnete Bereitstellungsdienst beinhaltet die grundlegende Bereitstellungsinfratruktur für Eclipse und alle Equinox-basierenden Applikationen. Neben diesen Komponenten, existieren weitere wie der *Incubator, Security* und *Server*.[7] Dieser Aufbau verdeutlicht das es sich bei Equinox um eine typische komponentenbasierte Plattform handelt. Im Laufe der Arbeit ist zu erkennen, dass sich diese Struktur im Eclipse RCP-Framework 4.1 wieder findet.

Das Equinox Framework stellt damit das Grundgerüst für die Ausführung von RCP-Applikationen dar. Eine der wesentlichen Aufgaben des Frameworks ist es, benötigte Bundles zu starten und nicht benötigte wieder zu beenden. Dies ist für einen schnellen Programmstart und für eine effiziente Speichernutzung der Applikation unerlässlich.

2.3 Eclipse RCP-Framework

Basierend auf Equinox bietet das Eclipse RCP-Framework die Möglichkeit der Entwicklung von Applikationen auf einer *Rich Client Plattform*(RCP). Der Rich Client ist dabei als eine Weiterentwicklung des Fat Clients zu betrachten. Beide zeichnen sich durch die Verarbeitung der Daten auf der Seite des Clients aus und stellen den Gegensatz zum Thin Client dar. Im Unterschied zum Fat Client sind Rich Client Anwendung durch eine starke Modularisierung einfacher zu verteilen und zu aktualisieren. Diese Module oder Komponenten werden im RCP-Umfeld auch als *Plug-ins* bezeichnet. Im Allgemeinen erfüllen den gleichen Zweck wie die Bundles im OSGi Umfeld. Im Rahmen dieser Arbeit werden sie daher als Synonyme betrachtet. Bevor genauer auf das Eclipse RCP-Framework eingegangen wird, illustriert die Abbildung 2.3 in Form eines

[5] Durch einen einzelnen Prozess, wird ein zunehmend komplexeres System erzeugt. Das System startet sich gewissermaßen selbst.

[6] `http://eclipse.org/equinox/bundles/`

[7] `http://eclipse.org/equinox/`

Venn-Diagramms die Einordnung aller bisher erörterter Begriffe. Von außen nach innen zu interpretieren, stellt OSGi eine Spezifikation eines möglichen Frameworks dar. Das Equinox Framework ist wiederum eine mögliche Umsetzung der OSGi Spezifikation. Das im Folgenden beschriebene Eclipse RCP-Framework konkretisiert die Anwendung von Equinox.

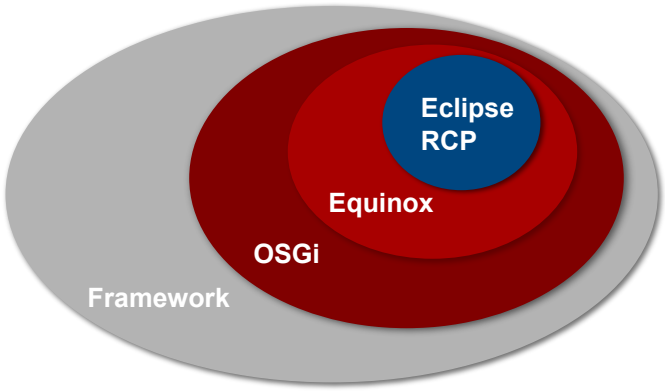

Abbildung 2.3: Einordnung der bisherigen Begriffe

Grundlegend besteht eine RCP-Applikation aus Plug-ins. Aus dieser Sichtweise ist die Eclipse IDE, als prominentes Beispiel einer RCP-Anwendung, eine mögliche Ansammlung verschiedener Plug-ins. Abbildung 2.4 verdeutlicht diesen Architekturgedanken. Basierend auf der Equinox Runtime, bilden das `SWT`- und das `JFace` Plug-in die wesentlichen GUI-Bestandteile, der Anwendung. Als Workbench soll an dieser Stelle das Arbeitsumfeld der Applikation verstanden werden. Konkreter wird auf diese Komponente im Abschnitt 3.2 eingegangen. Es sei drauf hingewiesen, dass die in Abbildung 2.4 dargestellten

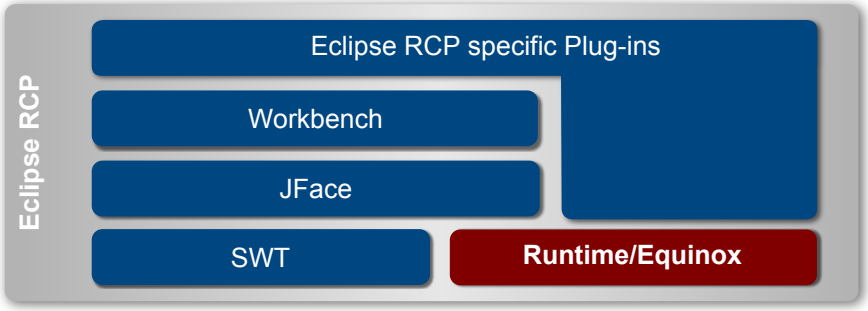

Abbildung 2.4: Architektur der Eclipse Rich Client Platform

Plug-ins, sich analog den *Equinox Bundles* in Abbildung 2.2 in weitere Plug-ins aufteilen, von denen eine RCP-Applikation in der Regel auch nur einen

Teil nutzt. Um bestimmte Funktionalitäten in der Anwendung umzusetzen, werden spezifische Plug-ins benutzt. Im Falle der Eclipse IDE wären das beispielsweise Editoren oder Debugfunktionen. Das Nutzen der Funktionen wird über *Depedencys*(Abhängigkeiten) und *Extension Points* gesteuert. Während Dependencys zu anderen Plug-ins die Funktionalität des eigenen Plug-ins erweitern, bieten Extension Points die Möglichkeit die eigenen Funktionen zur Verfügung zu stellen.

Die Abbildung 2.5 zeigt einen beschrifteten Screenshot der Eclipse IDE. Jeder der hervorgehobenen Bestandteile, vor allem das Editor-View Konzept, ist typisch für eine Eclipse RCP-Anwendungen.

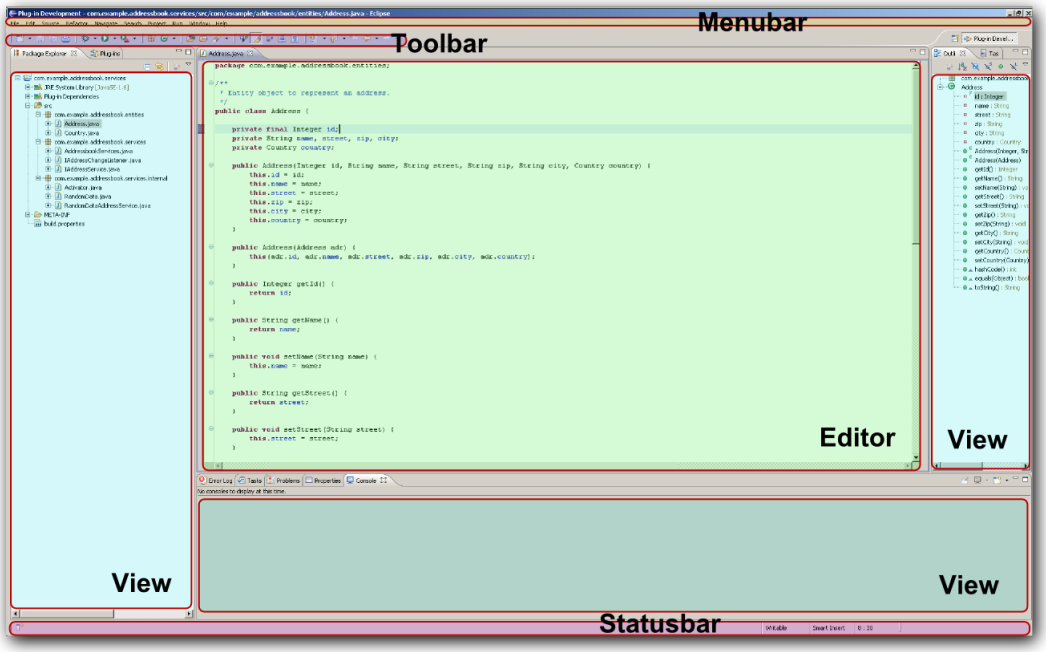

Abbildung 2.5: Screenshot der Eclipse IDE mit den typischen Bestandteilen einer Eclipse RCP-Applikation

Eine Anordnung von Editoren und Views wird als Perspektive bezeichnet. Diesem Prinzip geschuldet, sind die Gestaltungsmöglichkeiten relativ beschränkt und die GUI der Applikationen hängt gleicher Charakter an. Dieses starre Modell aus Editoren, Views und Perspektiven ist zudem nur mit hohem Aufwand erweiterbar. Auch der Vorteil der Modularisierung kann sich als Nachteil herausstellen. So gibt es durch eine fehlende zentrale Instanz keine Möglichkeit für den Anwendungsentwickler an zentraler Stelle Änderungen vorzunehmen.

Um diesen Nachteilen zu begegnen wurde im Juli 2009 damit begonnen unter dem Thema *e4 ist die nächste Generation der Entwicklungsplattform für eclipse-basierte Applikationen*[8] Technologien zu entwickeln. Diese sollten die Entwicklung von Komponenten vereinfachen und deren Wiederverwendung sowie Anpassung in einer Vielzahl von Applikationen und Umgebungen verbessern. Diese Ziele sollten laut [Athor09] durch verschiedene Lösungsansätze erreicht werden:

- Ein Service-orientiertes Programmiermodell, auf Grundlage von OSGi, welches eine bessere Abgrenzung von Softwarekomponenten von deren Umgebung bietet.
- Die GUI ist als einheitliches Modell dargestellt. Dieses Modell kann generell bearbeitet und erweitert werden, so dass für die Entwicklung und Anpassung wenig oder keine Programmieraufwand nötig ist.
- Die Benutzung von Web Styling Technologien (wie CSS), die es erlaubt das Aussehen der Benutzeroberfläche grenzenlos anzupassen, ohne den Programmcode anpassen zu müssen.

Im Juli 2010 wurde mit Eclipse 4.0 die erste Version der neuen Entwicklungsumgebung veröffentlicht in der die neuen Technologien zum Einsatz kamen. Die weiterentwickelte Version 4.1 erschien im Juni 2011.

[8] Vgl. [e4Evang]

3

Evaluierung des Eclipse RCP-Framework 4.1

Dieses Kapitel geht umfassend auf die Merkmale der Entwicklung mit dem Eclipse Framework 4.1 ein. Zu Beginn des Kapitels werden die grundlegenden Modelle beschrieben. Dies dient dem besseren Verständnis für die im Anschluss beschriebenen Entwicklungsansätze. Zur Beantwortung der Fragen in der Zielsetzung der Arbeit, umfasst das Kapitel alle für die Evaluierung relevanten Entwicklungsaspekte des Frameworks. Das Finden geeigneter Kriterien und die Einschätzung der Produktionstauglichkeit auf Grundlage dieser, rundet das Kapitel ab. Anzumerken ist, dass die verwendeten Code-Listings, zur Verdeutlichung der Entwicklungsansätze, Vereinfachungen darstellen und daher auf Importanweisungen oder Ähnliches der Übersichtlichkeit halber verzichtet wird.

3.1 Begrifflichkeiten: e4 vs. Eclipse 4.x

Für ein gemeinsames Verständnis wird auf die Begrifflichkeiten e4, Eclipse 4.x und Eclipse 4.1 Framework eingegangen. Wie im vorhergehenden Kapitel angedeutet, bezeichnet der Begriff *e4* einen Ort an dem neue Technologien für Eclipse entwickelt werden [Athor09]. e4 wird auch als Inkubator, gewissermaßen als Technologie-Brutstätte, bezeichnet. e4 versteht sich laut [e4Evang] auch als *RCP 2.0*. Wesentlich ist, dass sich die entwickelten Technologien nicht ausschließlich in der Version 4.x verwenden lassen, sondern ebenfalls in der aktuellen Version 3.x. Beispiele dafür sind der CSS Support zur Gestaltung der GUI und die Dependency Injection, zur Reduzierung der Abhängigkeiten zwischen Komponenten.

Eclipse 4.x oder das *Eclipse 4.1 SDK* bezeichnet die Entwicklungsumgung. Diese wiederum basiert auf den e4-Technologien.

Das *Eclipse 4.1 Framework* als Rahmenwerkzeug der Entwicklung umfasst sowohl e4 als auch das Eclipse 4.1 SDK. Daraus resultierend sind e4-Technologien und die Entwicklungsumgebung, als Anwendung der Technologien, Gegenstand der Evaluierung.

Ist im weiteren Verlauf der Arbeit von *e4-Applikationen* die Rede, bezeichnet dies immer die Entwicklung von Eclipse RCP-Anwendungen auf Grundlage der e4 Technologien mit dem Eclipse 4.1 SDK.

3.2 Applikationsmodell: Workbench

Ein zentraler Bestandteil des Eclipse RCP-Frameworks und der daraus resultierenden Anwendungen ist das *Workbench Modell* oder kurz die *Workbench*. Die Workbench beschreibt ein abstraktes Arbeitsumfeld. Es handelt sich um einen leeren Applikationsrahmen, der mit Plug-ins konkretisiert werden muss. Dieser Rahmen besteht dabei aus grafischen, wie einem Fenster zur Platzierung von Widgets, und aus nicht grafischen Komponenten, wie Commands. Das Modell der Workbench wird in der Version 4.x durch ein EMF-Modell repräsentiert, d.h. das e4-Workbench Modell basiert auf dem Eclipse Modeling Framework (kurz EMF), auf welches im folgenden Abschnitt näher eingegangen wird.

3.2.1 e4-Workbench Modell

Die e4-Workbench ist in Form von EMF-Modellobjekten aufgebaut. Mit Hilfe des Eclipse Modeling Frameworks wurden alle Elemente, die eine e4-Applikation ausmachen, modelliert. Das EMF wurde von den e4 Entwicklern gewählt, weil mit diesem die Serialisierung von Objekten oder dem ganzen Modell möglich ist, es ein Eventsystem bietet, welches über Zustandsänderungen informiert und es zudem wohldefinierte Möglichkeiten zur Erweiterung bietet. Laut [Vogella10] zeichnet es sich zudem durch eine schlanke Laufzeitumgebung und hochgradige Optimierung aus.
Die e4-Workbench verfolgt im Vergleich zur Eclipse Version 3.x einen integrativen Ansatz [SchiBo10]. Realisiert wird der Ansatz durch das zentrale Applikationsmodell. Dieses vereint UI-Elemente,, wie Fenster und Menüs und nicht visuelle Komponenten wie Commands und Handler. Dem Entwickler steht mit dem e4-Workbench Modell eine zentrale Programmstruktur zur Verfügung, in der die vollständige Anwendung, auch zur Laufzeit, eingesehen und beeinflusst werden kann. Die Abbildung 3.1 stellt dieses Modell dar.

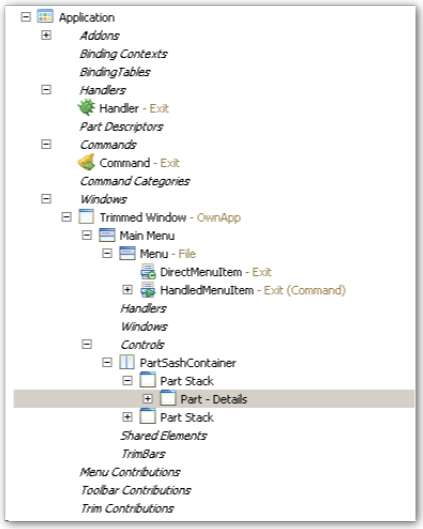

Abbildung 3.1: Ansicht des e4-Workbench Modells im e4-Worbench Modelleditor im Eclipse 4.1 SDK

Die bereits angesprochenen, visuellen Komponenten des Modells in der Abbildung 3.1 sind das Trimmed Window mit Menu als Fenster der Applikation und der PartSashContainer mit PartStacks und Parts als Inhalt des Fensters. Nicht visuelle Komponenten sind die Addons, Handler und Commands. Eine ausführliche Beschreibung der Arten von Komponenten ist im späteren Teil des Kapitels unter 3.4.3 zu finden.

Aufgaben des Workbench Modells

Die wesentlichen Aufgaben der Workbench untergliedern sich nach [SchiBo10] in drei Teile. Eine Aufgabe ist die *Abbildung der UI-Struktur* der laufenden Applikation. Das bedeutet, dass sich das Modell während der Laufzeit in ständiger Veränderung befindet. Es ändern sich Elemente, in ihrer Größe und/oder ihrer Position, oder werden hinzugefügt/entfernt. Die sichtbaren Widgets spiegeln damit immer den aktuellen Status des Workbench Modells wieder. Dieses Verhalten wird als *Live*-Modell bezeichnet. Erwähnenswert ist dabei, dass das Workbench Modell die UI-Elemente abstrakt abbildet. Das konkrete Umsetzen der Widgets liegt auf der Seite der Rendering-Engine[1]. Das Modell bietet nur die Integrationsplattform der anwendungsspezifischen UI-Contributions und zielt nicht darauf deren Inhalt abzubilden. Das Modell endet dort, wo anwendungsspezifischer Inhalt anfängt. Die UI-Elemente lassen sich weiterhin in 2 Teile unterteilen:

- Container-Elemente: Aufgabe der Containerelemente ist es, eine Baumstruktur zu ermöglichen, in welcher die zweite Art von Elementen Platz findet. Die wichtigsten Container sind davon PartSash und PartStack. Die Sash Komponente dient dazu mehrere Parts gleichzeitig anzuzeigen, wogegen der Stack immer nur einen Part zur gleichen Zeit anzeigt.

- Item-Elemente: Diese Elemente bilden die Blätter vom Strukturbaum des Modells. Typische Vertreter sind MenuItem oder Part.

Die zweite Aufgabe ist die *Einbettung von UI-Contribution*. Um die anwendungsspezifischen Elemente, z.B. den Inhalt von Parts, abzubilden, werden Plain Old Java Objects (POJOs) verwendet. Daher ist die zweite wichtige Aufgabe des Modells, diese POJOs in Beziehung mit den UI-Strukturelementen zu bringen. Das Attribut, welches dafür die größte Bedeutung hat, trägt den Namen URI. Das typische Format einer solchen URI ist platform:/plugin/de.preinhol.example.e4/ de.preinhol.example.e4.parts.DetailPart. Damit beinhaltet die URI Name der Klasse und die Angabe aus welchen OSGi-Bundle diese geladen werden soll.
Drittes Aufgabenfeld ist die *Speicherung und Wiederherstellung des Applikationsstatus* nach einem Neustart der Anwendung.

Das e4 Workbench Modell setzt sich aus drei Teilen zusammen. Zum einen aus dem bisher beschriebenen Applikationsmodell, welches den Hauptbestandteil

[1] Nähere Erläuterungen zum Renderprinzip in e4-Applikationen siehe 3.4.8

repräsentiert. Weiterhin besteht die Möglichkeit dieses Modell durch XMI-Datei oder Code zu erweitern.[2] Anpassungen durch den Nutzer werden ebenfalls registriert und gespeichert. Diese Änderungen werden als *Deltas* bezeichnet und befinden sich in einer XML-Datei mit gleichen Namen im Workspace der Applikation.[3] Sollen die Änderungen am Modell, z.B. das Verschieben eines Parts, nicht gesichert werden, muss der Parameter *clearPersistedState* beim Start angegeben werden.[4]
Das e4-Worbenchmodell kombiniert diese drei möglichen Teile bei jedem Start der Applikation. Anhand der Abbildung 3.2 wird dieser Vorgang verdeutlicht.

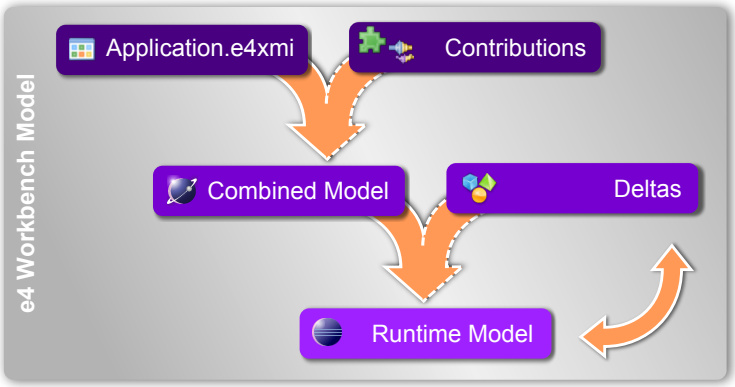

Abbildung 3.2: Die Erzeugung des Laufzeitmodells des e4-Workbench Models.

Die Applikation ist in der *Application.e4xmi* als Grundmodell festgelegt. Die oben beschriebenen visuellen und nicht visuellen Elemente werden immer an dieser Stelle definiert. Zusätzliche Plug-ins werden über Extension-Points bzw. wie oben angedeutet mittels XMI-Dateien oder Code dem Modell hinzugefügt. Das Grundmodell ergibt zusammen mit dem optionalen Erweiterungen das *Combined Model*. Die als Delta bezeichneten Laufzeitveränderungen werden im Anschluss auf das Combined Model angewendet. Das entstehende Modell ist das tatsächliche *Runtime Model* der Applikation. Während der Laufzeit werden Veränderungen, etwa das Verschieben eines Parts, in die Deltas übertragen. Änderungen ohne Auswirkung oder Effekt werden ignoriert (das Schließen und erneutes Öffnen einer Komponente). Die Änderung der Komponenten erfolgt dabei über die jeweiligen IDs. Wie bereits erwähnt, kann der Delta-Mechanismus durch Parameterangaben abgeschaltet werden.

[2]Diese Erweiterungen werden auch Contributions genannt, mehr dazu unter 3.4.6.
[3]Konkret unter: .metadata/.plugins/org.eclipse.e4.workbench/deltas.xml.
[4]Dies entspricht in etwa dem configurer.setSaveAndRestore(false) Flag in Eclipse 3.x.

3.2.2 Zusammenfassung

Das neue e4-Workbench Modell bietet im Vergleich zur Version 3.x die folgende Vorteile:

- Das Modell ist die zentrale Komponente der Applikation.
- Das Modell ist durch den Workbench Modelleditor übersichtlich dargestellt und kann in diesem leicht angepasst werden.
- Das Modell ist durch Anpassungen zur Laufzeit sehr flexibel.
- Es gibt keine Unterscheidung zwischen View und Editor, beide werden als Parts zusammengeführt.
- Perspektiven sind optional.
- Das Modell ist für eigene Funktionen, auf Grundlage des EMF, leicht erweiterbar.

Das Modell hat jedoch die folgenden Grenzen:

- Es wird nur der Applikationsrahmen modelliert.
- POJOs sind nötig, um den Rahmen der Applikation mit Inhalt zu füllen.
- Dieser Inhalt ist nicht Teil des e4-Workbench Modells.
- Der Inhalt immer über URIs bekannt gemacht werden.

3.3 Programmiermodell: Dependency Injection

Die Grundlage des Programmiermodells bilden die in Kapitel 2 definierten Frameworks und das dort erwähnte das Paradigma der *Inversion of Control*. Dieses beschreibt die Umkehrung der Steuerung. Eine Funktion der Applikation wird beim Framework registriert und zu späterem Zeitpunkt aufgerufen. Die Anwendung selbst steuert nicht den Kontrollfluss, sondern nutzt Standardfunktionen und gibt damit die Steuerung der Anwendung an das Framework ab. Dieses Funktionsmerkmal ist das Unterscheidungskriterium zwischen Frameworks und Programmbibliotheken. Bibliotheken fassen lediglich Funktionen zusammen, die Kontrolle bleibt bei der Anwendung. Vorteil dieser Steuerungsübergabe ist, dass sich der Entwickler auf die Implementierung der Geschäftslogik konzentrieren kann. Ein typisches Beispiel zur Verdeutlichung sind *Listener* gemäß dem Observer-Muster. So wird der Programmfluss im Falle eines Action-Listener, der an einen Button gebundenen wurde, abgeben. Eine Spezialform der Inversion of Control ist die *Dependency Injection*.

3.3.1 Grundlagen der Dependecy Injection

Die Dependency Injection ist ein Entwurfsmuster und dient in Softwaresystemen dazu die Abhängigkeiten zwischen Komponenten und Objekten zu minimieren. Eine Übersetzung, etwa in Abhängigkeitsinjizierung ist unüblich, stattdessen wird von Dependency Injection oder kurz DI gesprochen. Der Begriff an sich geht dabei auf M.Fowler zurück.[5] Das Muster bezieht sich nur auf die Erzeugung und Initialisierung von Objekten. Bevor näher auf das Prinzip der DI eingegangen wird, soll zunächst das Problem erläutert werden, welches mittels DI gelöst werden soll. In einem klassischen objektorientieren System ist jedes Objekt selbst zuständig, sich die benötigten Abhängigkeiten von Objekten und Ressourcen zu verschaffen. Dazu muss das Objekt Wissen über seine Umgebung beinhalten, die es für die Erfüllung der eigentlichen Aufgabe nicht benötigt. Insbesondere muss es die entsprechenden Objekte erzeugen können, um deren konkrete Implementierung zu kennen. Das einfache Beispiel einer Pizza-Belag-Beziehung soll dies illustrieren.

```
1 public interface IPizza{
2  ... }
3
4 public interface ITopping{
5  ... }
```

Listing 3.1: Interfaces Pizza Beispiel

[5] As a result [...] we settled on the name Dependency Injection. [Fowler04]

Nach einer Definition der Interfaces folgt die konkrete Implementierung.

```
1  public class DefaultToppingImpl implements ITopping {  ...  }
2
3  public class DefaultPizzaImpl implements IPizza {
4    private ITopping topping = new DefaultToppingImpl();
5    topping.add(...);
6  }
7
8  public class Application {
9    public static void main(String args[]) {
10     IPizza pizza = new DefaultPizzaImpl();
11     ...
12   }
13 }
```

Listing 3.2: Implementierung 1 Pizza Beispiel

In dieser Form der Implementierung ergeben sich einige Kritikpunkte. Die `DefaultPizzaImpl` hat eine direkte Abhänigkeit von der `DefaultToppingImpl` (Zeile 4). Würde für einen JUnit-Test statt der `DefaultToppingImpl` eine `DummyToppingImpl` verwendet werden, muss der Code angepasst werden. Grund ist die Erzeugung und Zubereitung des Belags innerhalb der `DefaultPizzaImpl`. Dies ist nicht die primäre Aufgabe dieser Klasse. Ein besserer Ansatz wäre es, die Abhängigkeit vom Belag in die Pizza zu injizieren. Folgendes Beispiel demonstriert die Verbesserung(Interfaces und DefaultToppingImpl bleiben unverändert).

```
1  public class DefaultPizzaImpl implements IPizza {
2    private ITopping topping;
3
4    public DefaultPizzaImpl(ITopping toppingImpl) {
5      topping = toppingImpl;
6    }
7  }
8
9  public class PizzaFactory() {
10   public static IPizza makePizza() {
11   IToppting topping = new DefaultToppingImpl();
12   topping.add(...);
13   ...
14   return new DefaultPizzaImpl(topping);
15   }
16 }
17
18 public class Application {
19   public static void main(String args[]) {
20     IPizza pizza = PizzaFactory.makePizza();
21     ...
22   }
23 }
```

Listing 3.3: Implementierung 2 Pizza Beispiel

In dieser Umsetzung bereitet die `PizzaFactory` die Pizza zu. Die Abhängigkeit geht damit von der `DefaultPizzaImpl` zur Factory-Klasse. Die DI ist in diesem Fall die Parameterübergabe im Konstruktor der Pizza-Implementierung. Sollte die Pizza mit einem anderen Belag zubereitet werden müssen, bleibt die konkrete Implementierung unverändert. Der Code innerhalb der Factory- Klasse müsste jedoch weiterhin angepasst werden. Eine Verbesserung des Factory-Prinzips bietet ein framework-unterstützter Ansatz der folgenden Art.

```
<service-point id="PizzaMakerService">
    <invoke-factory>
      <construct class="Pizza">
        <service>DefaultPizzaImpl</service>
        <service>DefaultToppingImpl</service>
      </construct>
    </invoke-factory>
</service-point>
```

Listing 3.4: Implementierung 3a Pizza Beispiel

Dieses Codebeispiel zeigt die mögliche Konfiguration eines Services in einer externen XML Datei, der folgende Ausschnitt zeigt die Verwendung.

```
public class Application {
  Service service = (Service)DependencyManager.get("PizzaMakerService");
  IPizza pizza = (IPizza)service.get(Pizza.class);
  System.out.println("Price of the pizza is"+ pizza.getPrice());
  }
}
```

Listing 3.5: Implementierung 3b Pizza Beispiel

Das Pizzaobjekt wird an dieser Stelle vom `PizzaMakerService` bereitgestellt. Dieser kümmert sich um die Erzeugung der konkreten Implementierungen. Ziel ist die Trennung der Service Konfiguration, von dessen Verwendung um eine lose Kopplung zu erreichen. Die DI wird zudem nach der Stelle unterschieden, an die Abhängigkeiten injiziert werden. So ist von Contructor-, Setter-, und Interface-Injection im Artikel von [Fowler04] die Rede.

Wie am Beispiel zu erkennen ist, kann bei der Dependency Injection von einer Verallgemeinerung der Fabrikmethode gesprochen werden. Es handelt sich weniger um ein neues Entwurfsmuster, sondern viel mehr um ein bekanntes und angepasstes Prinzip mit neuem Namen.

3.3.2 e4 Implementierung der Dependency Injection

Die Grundlage für die Umsetzung der Depencency Injection innerhalb von e4-Applikationen bildet der *Java Specification Request 330*[6]. Die Konfiguration der DI wird nicht in externe Dateien ausgelagert, sondern stattdessen mittels Java-Annotations im Quellcode vermerkt werden. Vorteil dieser Annotationen ist, dass zum Verständnis des Quellcodes keine externe Quelle benötigt wird. Jedoch wirken sich die zunehmende Anzahl an Annotations auf die Übersichtlichkeit des Quellcode aus. Objekte, welche injiziert werden sollen, müssen in der zentralen Komponente der DI registriert sein. Diese Komponente wird als *IEclipseContext* bezeichnet und umfasst sowohl Objekte, Konstanten und Services.[7] Die möglichen Verwendungen der Codeinjektion mittels @Inject wird mit den Anforderungen und kurzen Codebeispielen in der Tabelle 3.1 zusammengefasst.

Tabelle 3.1: Inject Verwendungsmöglichkeiten

Injection	Code Beispiel
@Inject <Konstruktor> - optional für öffenliche Standardkostruktoren - maximal 1 Konstruktor kann annotiert werden - keine oder mehrere Abhängigkeiten als Argumente	1 @Inject 2 public ExampleConstructor(Composite parent){ 3 // Composite-Objekt parent wird injiziert 4 }
@Inject <Feld> - nicht final - jeder gültige Name	1 @Inject 2 protected ExampleService service; 3 // das ExampleServiceobject wird injiziert
@Inject <Methode> - nicht abstract - jeder gültige Name - darf ein Ergebnis zurückgeben - deklarieren keine eigenen Typparameter - keine oder mehrere Abhängigkeiten als Argumente - @Inject ignoriert das Ergebnis, aber nicht-void Ergebnisse erlauben Methoden in anderen Kontext	1 @Inject 2 private RetVal doSomething(InjectVal value){ 3 // InjectVal value wird injiziert 4 // return-Value steht im IEclipseContext 5 // zur Verfuegung? 6 }

Die @Inject Annotation kann auf statische sowie Instanz Felder angewendet werden, auch der Zugriffsmodifikator ist beliebig. Die Reihenfolge der Injek-

[6] Vgl. [JSR330]

[7] Auf Details der Komponente wird unter 3.4.2 eingegangen

tion innerhalb einer Klasse entspricht jener in der Tabelle: `Konstruktoren`, `Felder`, `Methoden`. Felder und Methoden in Superklassen werden vor denen der Subklassen injiziert. Die Reihenfolge der Injektion von Feldern und Methoden innerhalb der Klasse ist nicht festgelegt. Verändern sich die Werte im IEclipseContext, werden diese *re-injiziert*. Standardmäßig werden die injizierten Werte am Typ erkannt. Wenn jedoch mehrere Objekte eines Typs verfügbar sind, können diese anhand eines Bezeichners unterschieden werden. Nachfolgend sind die bedeutendsten Annotationen für die Entwicklung von e4-Applikationen aufgelistet. Codebeispiele finden sich im Anhang A.2

`@Named` Die Annotation ist ein stringbasierte Bezeichner in Verbindung mit @Inject. Sollte das bezeichnete Objekt nicht im Kontext gefunden werden, kommt es zu einer Exception. Daher wird diese in der Regel mit @Optional kombiniert.

`@Optional` Diese e4-spezfische Annotation kann genutzt werden um Methoden, Felder oder Parameter zu markieren. Wie erwähnt kommt es zu einem Fehler, wenn ein Wert, der injiziert werden soll, nicht gefunden wird. Ist dieser jedoch mit @Optional markiert, kommt es zu den folgenden Verhalten.

- Felder mit dieser Markierung werden nicht injiziert.
- Für Parameter wird ein `null` Wert wird injiziert.
- Methodenaufrufe mit dieser Annotation werden übersprungen.

`@PostConstruct` und `@PreDestroy` Die beiden Annotationen stehen für die Verwaltung des Lebenszyklus' zur Verfügung. Mit @PostConstruct annotierte Methoden, werden aufgerufen sobald alle Objekte vollständig injiziert wurden. @PreDestroy wird aufgerufen bevor diese aus dem Kontext gelöst werden. Listing A.8 zeigt die Verwendung.

`@Active` Diese Markierung von Parametern signalisiert, dass aktive Komponenten injiziert werden soll. Beispielsweise der aktuell aktive Part eines Fensters. Auch hier ist die Kombination mit @Optional empfohlen.

`@Preference` Diese Annotation dient als Schnittstelle mit dem Eclipse Preference Framework. Bei einer Veränderung der Preference wird diese aktualisiert, der auf den Bundle referierende Wert `node` ist optional. Eine Klasse kann zudem eine Benachrichtigung über eine Preference-Veränderung durch die Injektion in eine setter-Methode entgegennehmen.

@CanExecute und @Execute Diese Annotationen dienen dazu Methoden, die von einem Handler ausgeführt werden sollen, zu kennzeichnen. Ein Handler POJO muss mindestens über eine mit @Execute annotierte Methode verfügen. Die optinale @CanExecute sollte einen `boolean`-Wert zurückliefern.

@Focus Parts können mit dieser Annotation eine Methode markieren, die beim Erhalt des Fokus' aufgerufen werden soll. Listing A.8 zeigt die Verwendung.

@Persist Parts können mit dieser Annotation eine Methode markieren, die bei der *savePart()*-Methode des EPartService aufgerufen werden soll. Die Verwendung zeigt Listing A.8 auf.

@GroupUpdates Mithilfe dieser Annotation kann dem Framework signalisiert werden, dass Aktualisierungen stapelweise/blockweise erfolgen sollen.

@UIEventTopic Die Anntoation dient der Kommunikation. Mit Hilfe des IEventBrokers können Objekte anhand von UIEventTopics geworfen und unterschieden werden. [Schin11c] bezeichnet diese Möglichkeit als *The e4 way to communicate*.

Anzumerken ist, dass e4 nicht auf einen DI-Container-Framework, wie Spring[8] oder Google Guice[9] aufsetzt, sondern diesen selbst implementiert. Das Codebeispiel A.8 verdeutlicht das Programmiermodell von e4-basierenden Applikationen. POJOs erhalten ihre benötigten Abhängigkeiten mittels Dendency Injection und werden über die URI mit dem zentralen e4-Workbench Modell verknüpft.

[8] `http://www.springsource.com/`
[9] `http://code.google.com/p/google-guice/`

3.3.3 Zusammenfassung

Folgend seien die Vor- und Nachteile der Dependency Injection in Anlehnung an [Straub11] aufgeführt.

Voteile durch Dependency Injection

- Die DI schafft eine saubere und deklarative Architektur.
- Die Wiederverwendbarkeit von Komponenten (POJOs) und Services wird in besonderem Maße unterstützt.
- Bereits während der Entwicklung sind Module unabhängig von der Implementierung anderer benötigter Komponenten.
- Eine alternative Implementierung einer Komponente ist ohne erneutes Kompilieren verwendbar.
- Die Entkopplung der Systemteile wird durch die DI begünstigt.

Nachteile durch Dependency Injection

- Fehler durch die DI sind i.d.R. nicht trivial lösbar. Vor allem Tippfehler können schwer zu finden sein.
- Da Teile der Programmlogik *ausgelagert/abstrahiert* werden, ist die Wartung schwieriger und unübersichtlicher.
- Zum Verstehen des Codes ist zusätzliches Wissen nötig.

3.4 Entwicklung von e4-Applikationen

Nachdem die Abschnitte 3.2 und 3.3 auf die grundlegenden Modelle eingegangen sind, behandelt dieser Abschnitt den Aufbau, die Architektur und die Entwicklung von e4-Anwendungen. Dabei wird auf alle für die Evaluierung des Frameworks relevanten Aspekte eingegangen, um die Frage *Was kann das Framework?* beantworten zu können.

3.4.1 Grundaufbau von e4-Applikationen

Es existieren zwei der Möglichkeiten der Entwicklung von e4-Applikationen. Zum einen mit der Eclipse 4.x SDK und zum anderen mit Hilfe des Eclipse 3.x SDK und der 4.x RCP target platform. Da Letzeres kaum praktische Relevanz hat, wird folgend immer von der ersten Möglichkeit Gebrauch gemacht. Für einen effektiven Entwicklungsprozess sind die *Eclipse e4-Tools* unerlässlich.[10] Diese beinhalten Wizards und Editoren für einzelne Bestandteile einer e4-Applikation und werden im Folgenden als notwendige Voraussetzung zur Entwicklung von e4-Applikationen betrachtet.
Um an eine Minimalapplikation zu gelangen, kann mittels Wizard eine solche erzeugt werden.[11] Das generierte Projekt und die generierte Anwendung sind auf Abbildung 3.3 dargestellt.

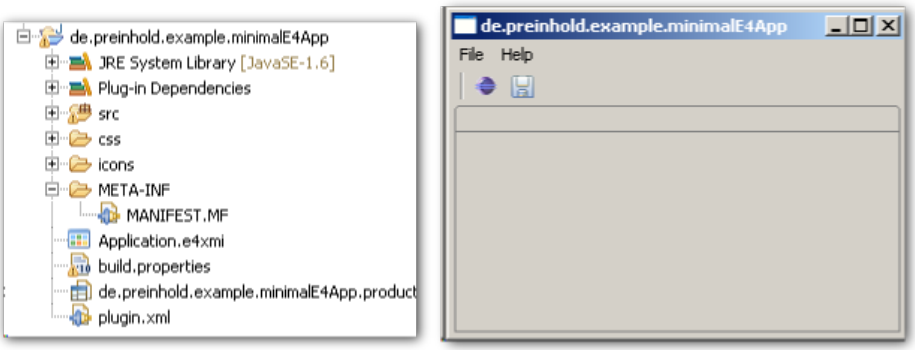

Abbildung 3.3: Projekt und Fenster einer minimalen 4.x RCP-Applikation

Die generierten Ordner *css* und *icons* sind dabei optional. Die folgende Seite geht auf die Mindestanforderungen einer e4-Applikation näher ein.

[10] Für Installationshinweise siehe Anhang A.1
[11] *New -> e4 -> E4 Application Project*

Application.e4xmi Jede e4-Anwendung muss über eine *.e4xmi* Datei, welche das Workbench Modell repräsentiert, verfügen. Der Name ist beliebig, jedoch hat sich die Standardbezeichnung *Application.e4xmi* etabliert. Eine e4-Applikation benötigt mindestens 6 Add-ons, welche in der Application.e4xmi angegeben werden müssen. Diese *Add-ons* ermöglichen grundlegende Funktionen und werden beim Erzeugen einer e4-Applikation mittels Wizard automatisch generiert. Die Add-ons umfassen dabei Command-, Context- und Bindingservices, die vollständigen Namen und die konkrete Verwendung sind im Anhang A.3 zu finden.

.product Eine jede e4-Applikation muss eine Produktkonfiguration definieren. Der Name des Produktes ist beliebig, als *Application* muss in der Produktdefinition jedoch *org.eclipse.e4.ui.workbench.swt.E4Application* angegeben werden. Beim Erzeugen einer e4-Applikation mittels Wizard, ist die Definition Bestandteil der generierten Dateien.

Dependencies Die aufgelisteten Abhängigkeiten sind Voraussetzung für das Funktionieren einer e4-Applikation.

- org.eclipse.core.runtime
- org.eclipse.e4.ui.workbenh
- org.eclipse.e4.core.services
- org.eclipse.e4.core.di
- org.eclipse.e4.core.contets
- org.eclipse.e4.ui.services
- org.eclipse.e4.ui.workbench.swt
- org.eclipse.e4.ui.css.swt.theme
- javax.inject
- javax.annotation
- org.eclipse.equinox.ds
- org.eclipse.equinox.event
- org.eclipse.e4.ui.workbench.renderers.swt

Extensions Beim Erzeugen eines Produktes wird automatisch der Extension Point *org.eclipse.core.runtime.product* in der plugin.xml angelegt. Innerhalb des Produkt-Extension Points muss eine *property*, mit dem Namen *applicationXMI* und dem Value *Pfad der Application.e4xmi* angelegt sein. Der Pfadname setzt ich aus Paketname und mit Slash getrenntem Namen der Application.e4xmi zusammen. Im Falle der in Abbildung 3.3 generierten Applikation *de.preinhold.example.minmalE4App/Application.e4xmi*.

3.4.2 Architektur von e4-Applikationen

Aufbauend auf dem Betriebssystem und der Java VM bilden EMF, SWT und JFace sowie die Equinox Runtime die Grundlage für die *Eclipse 4.x Application Platform*, kurz EAP [Schin11a] oder E4AP [e4DiWiki]. Diese Plattform stellt mit dem Applikations- und Programmiermodell die Grundlage einer jeden e4-Applikation dar. Dennoch bietet die E4AP technologische Freiheit in Hinsicht des Rendermodells, so ist es möglich eine e4-Applikation mit jeder nativen UI-Technologie zu entwickeln [Schin11b]. Im Standardfall wird dabei auf die SWT-Bibliothek zurückgegriffen.

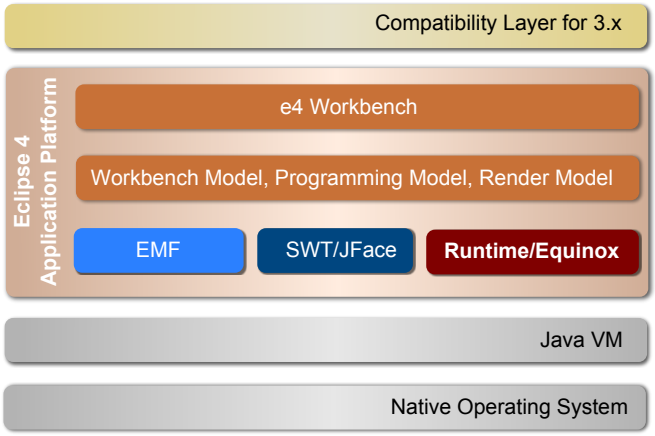

Abbildung 3.4: Die Architektur einer 4.x RCP-Applikation

Ein wichtiger Bestandteil der Architektur stellt der Kompatibilitätslayer dar. Dieser dient dazu Plug-ins aus der Version 3.x in die E4AP zu integrieren. Damit ist es zum einen möglich vollständige mit 3.x entwickelte Anwendungen in der 4.x Umgebung zu betreiben. Ein weiterer wichtiger Punkt ist aber die Integration von Standardkomponenten wie Editoren oder Wizards, die in Version 4.1 noch nicht portiert worden. Aus diesem Grund finden die 3.x Komponenten weiterhin Verwendung [CompLa]. Ohne diesen Layer ist eine e4-Applikation, wie die Eclipse 4.1 SDK, nicht lauffähig.
Um eine 3.x Applikation auf der EAP erfolgreich zu starten, ist es laut [BugCo] lediglich nötig, die Dependencys zu *org.eclipse.e4.ui.workbench.addons.swt*, zu *org.eclipse.equinox.ds*, zu *org.eclipse.equinox.event* und zu *org.eclipse.equinox.util* herzustellen. Tests mit Eclipse 4.1 und der generierbaren 3.x Mail RCP-Anwendung bestätigten dies. Wie [BugCo] zu entnehmen ist, wird die notwendige .e4xmi Datei in Form der LegacyIDE.e4xmi standardmäßig bereitgestellt und ermöglicht so das Starten der 3.x Applikation im 4.x Umfeld.

3.4.3 IEclipseContext – DI Zentrale der e4-Applikationen

Um die Dependency Injection zu ermöglichen, besitzt das Eclipse RCP-Framework als zentrale Komponente den *IEclipseContext*. In diesem befinden sich Objekte und Services, auf die mittels den Annotationen @Inject oder @Named zugegriffen werden kann. Der IEclipseContext ist hierarchisch aufgebaut und vergleichbar mit einer *Map* der Form `Map<String, Object>`. Die Abbildung 3.5 verdeutlicht den internen Aufbau.

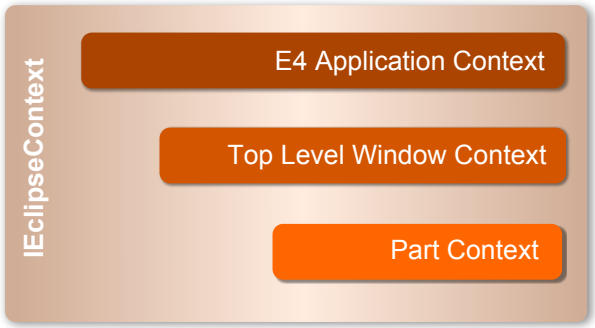

Abbildung 3.5: Der Aufbau des IEclipseContext

An oberer Stelle der Hierarchie befindet sich der *Application Context*. Dieser beinhaltet wesentliche Parameter und Services einer e4-Applikation, wie Zugriff auf das Workbench Modell oder Services, welche Command oder Renderaufgaben übernehmen. Der darunter liegende *Top Level Window Context* umfasst den Zugriff auf Modellobjekte wie Tastenzuweisungen oder UIElemente, zudem Services die Zugriff auf aktive oder ausgewählte, grafische Komponenten liefern. Die untere Stufe der IEclipseContext-Hierarchie beschreibt den *Part Context*, das heißt den Zugriff auf die Modellkomponenten, die in diesem Part benutzt werden. Eine vollständige Auflistung der vorhandenen Services in den Schichten des IEclipseContext findet sich im Anhang A.4
Nach der folgenden Strategie wird nach einem angeforderten Objekt im Kontext gesucht. Das Framework sucht zunächst nach einem Objekt in der unmittelbaren Umgebung in der aktuellen Context-Schicht. Sollte das passende Objekt nicht gefunden werden, wird in der darüber liegenden Schicht gesucht. Kann das Objekt nicht im Kontext gefunden werden, wird versucht es als OSGi Service zu finden. Für Letzteres ist die Klasse *OSGiContextStrategy* zuständig. Ist es nicht möglich das geforderte Objekt zu finden, kommt es zu einer `org.eclipse.e4.core.di.InjectionException`.
Neben dem extrahieren von Objekten aus dem IEclipseContext ist das Hinzufügen von Objekten zum Kontext interessant. Um etwa ein POJO eines Parts

MyPart hinzuzufügen, kann nach der folgenden Art verfahren werden.

```
1 @Inject
2 private IEclipseContext iEclipseContext;
3 //...
4 ContextInjectionFactory.make(MyPart.class, iEclipseContext);
```

Listing 3.6: Implementierung 1 IEclipseContext

Der Kontext muss dabei zu Verfügung stehen, dieser könnte wie im oberen Code Beispiel injiziert werden. Eine andere Möglichkeit an den IEclipseContext zu gelangen ist über den OSGi Standard.

```
1   // Hole Bundle Information
2   Bundle bundle = FrameworkUtil.getBundle(getClass());
3   BundleContext bundleContext = bundle.getBundleContext();
4   IEclipseContext iEclipseContext =
5       EclipseContextFactory.getServiceContext(bundleContext);
6
7   // ...
8
9   // Erzeuge Klasseninstanz im Kontext
10  ContextInjectionFactory.make(MyPart.class, iEclipseContext);
```

Listing 3.7: Implementierung 2 IEclipseContext

Der IEclipseContext als zentrale Komponente trägt im besonderen Maße bei das Prinzip der Inversion of Control umzusetzen. Neben der Regelung der Verfügbarkeit von Services und der Re-Injizierung geänderter Werte, stehen den Services Informationen zum Lifecycle zur Verfügung. Etwa um Clean-Ups durchzuführen, wenn der Service von keinem Client mehr verwendet wird [E4Ctx]. Um den Kontext bereits bei Start des Programms Informationen zukommen zu lassen, stehen innerhalb des e4-Applikationsmodells sogenannte Add-on Komponenten zur Verfügung.

3.4.4 Application.e4xmi – Modell der e4-Applikationen

Die Application.e4xmi stellt als Grundmodell der e4-Workbench einen wichtigen Bestandteil einer e4-Applikation dar. Wie Abbildung 3.3 zu erkennen gibt, befindet sich diese Modelldatei direkt im Verzeichnis des Plug-ins. Mit Hilfe des e4-Workbench Modelleditors, kann das Modell bearbeitet werden. Dieser öffnet sich mit installierten *Eclipse e4-Tools*[12] beim Doppelklick auf das Modell automatisch.

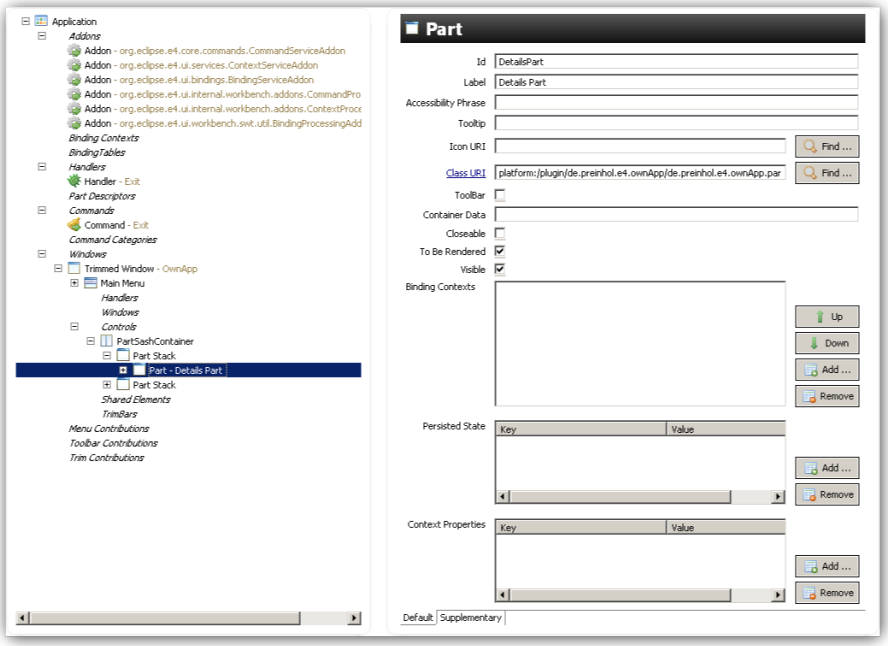

Abbildung 3.6: Der e4-Workbench Modelleditor

Der Editor ist zweigeteilt. Der linke Teil stellt als Baumstruktur das Workbench Modell dar, während der rechte Teil zur Eingabe konkreter Daten dient. Wie Abbildung 3.6 detaillierter darstellt, ist das Modell in zahlreiche Komponenten unterteilt. Diese sind sowohl grafische, wie nicht grafische Komponenten. Im rechten Teil des Editors sind besonders das *Id* und das *ClassURI* Feld von Bedeutung. Diese stellen die Verknüpfung zwischen eigenem Java-Programmcode und dem Workbench Modell dar. Die Beschreibung aller wichtigen Komponenten findet sich auf den folgenden Seiten.

[12] Für Installationshinweise siehe Anhang A.1

Add-ons

Die Add-onobjekte werden vom Framework innerhalb des IEContexts instantiiert. Diese sind global verfügbar und finden sich laut [E4Add] im Application Context wieder. Eine Standard e4-Applikation benötigt mindestens die unter 3.4.1 angesprochenen 6 Add-ons. Wichtig ist zu bemerken, dass die Add-onobjekte erzeugt werden, bevor die Renderengine das Applikationsmodell rendert. Add-ons können daher genutzt werden, um die Benutzeroberfläche anzupassen. Das im Anhang zu findende Min/Max Add-on erlaubt es Parts zu minimieren bzw. maximieren. Es passt hierbei die Erscheinung der MPartstacks an, um kleine min/max Button in den Ecken anzuzeigen.

Binding Contexts

Mit Hilfe der *Binding Contexts*, ist es möglich Tastenweisungen zu organisieren. Ein erzeugter Kontext lässt sich in weitere Gruppen unterteilen. So ist es beispielsweise möglich einen Kontext für Fenster und Dialoge zu erzeugen, in denen gemeinsame Tastenzuweisungen festgelegt werden. Dieser kann dann wiederum in einen Fensterkontext und einen Dialogkontext unterteilt werden, für welche spezifische Zuweisungen gelten. Die Binding Contexts werden für die Binding-Tables zwingend benötigt.

Binding Tables

Binding Tables fassen konkrete Tastenzuweisungen in Tabellen zusammen. Eine jede Tabelle benötigt dabei eine *Context Id*, die den Gültigkeitsbereich in Form eines Binding Context festlegt. Einer solchen Tabelle werden dann die Tastenzuweisungen zugeordnet. Ein *KeyBinding* benötigt neben der Tastensequenz einen *Command*, also einen Befehl, der beim Druck der Tastenkombination ausgeführt werden soll.

Handler

Ein *Handler* stellt die Implementierung eines bestimmten Verhaltes dar und kann an einen Command gebunden werden. Ein Handler ist mit einem POJO verknüpft. Anhand der Annotationen @Execute und @CanExecute, erkennt das Framework welche Methode ausgeführt werden soll. Während die erste Annotation notwendig ist, ist die zweite optional. In beiden Fällen werden die Parameter der annotierten Methoden aus dem IEclispeContext injiziert. Folgender Codeausschnitt soll die Erzeugung anhand eines Exit-Handlers verdeutlichen.

```
1  public class ExitHandler {
2
3   @Execute
4   public void execute(IWorkbench workbench) {
5     workbench.close();
6   }
7
8   @CanExecute
9   public boolean canExecute() {
10    if(checkAllConditions()) {
11      // ...
12    }
13  }
14 }
```

Listing 3.8: Implementierung ExitHandler

Üblicherweise werden alle Handler in einem eigenen **.handlers* Package gesammelt abgelegt. Optional kann der Handler einem Command zugewiesen werden. Die Id ist dann für eine Referenzierung obligatorisch.

Part Descriptors

Part Descriptors stellen abstrakte Parts dar, die mit Hilfe des im IEclipse-Context verfügbaren Part-Service instantiiert werden. Dazu muss dem Service die ID des Part Descriptors übergebenen werden, womit dieses Feld im Application Modell zwingend notwendig wird. Die Descriptors bieten neben der einfacheren Erzeugung den Vorteil einer übersichtlichen Auflistung vorhandener (abstrakter) Parts innerhalb des Applikationsmodells.

Commands

Ein Command ist eine Abstraktion für eine generische Aktion (speichern, öffnen, etc.) und stellt keine Implementierung eines Verhaltens dar. Dazu dienen die beschriebenen Handler. Der Command kann in Toolbars oder Menüs verwendet werden. Die Id des Commands darf nicht leer sein. Jedem Command können mehrere KeyBindings zugeordnet sein, aber nur genau ein Handler.

Command Categories

Ähnlich den Binding Contexts bieten die *Command Categories* die Möglichkeit der Kategorisierung und damit die bessere Verwaltung bei zahlreichen Commands. Im Gegensatz zu dem Binding Context sind die Command Categories jedoch optional.

Windows

Windows umfasst die sichtbaren Komponenten des Applikationsmodells. Es beinhaltet ein *TrimmedWindow* oder ein *Window*. Als TrimmedWindow wird ein Fenster bezeichnet, dass mit zusätzlichen Leisten (Toolbars) *ausgeschmückt* werden kann. Eine TrimmedWindow Komponente umfasst die folgenden Untergruppen:

Main Menu Dieser Unterpunkt ist optional und wird im Editorfeld der Elternkomponenten(Window, Part) über die CheckBox *Main Menu* ein- und ausgeschaltet. Das Main Menu entspricht dabei der MenuBar aus Abbildung 2.5. Innerhalb des Main Menüs können weitere Menüs mit Untermenüs und *MenuItems* angelegt werden. Als MenuItems zählen an dieser Stelle *HandledMenuItems*, *DirectMenuItems* und Separators. Wobei letzte als Trennkomponenten nur bedingt dazugehören. HandledMenuItems verlangen, wie der Name vermuten lässt, keinen Handler, sondern einen Command als aktive Komponente. DirectMenuItems benötigen dagegen *direkt* eine Handlerklasse. HandledMenuItems haben den Vorteil, dass automatisch Tastenkürzel an die Menukomponenten angehangen werden, wenn der dazugehörige Command eine Tastenzuweisung besitzt.

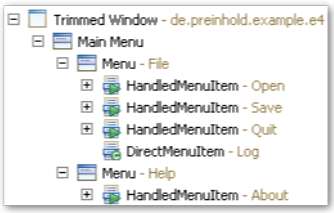

Abbildung 3.7: MenuItems im Editor des Applikationsmodells

(local) Handler Diese bieten die Möglichkeit lokal gültige Handler innerhalb des Fensters zu erzeugen.

Window Einem Fenster können weitere Fenster zugeordnet werden. Erzeugte Fenster in diesem Unterpunkt haben exakt den hier beschriebenen Aufbau an Unterpunkten.

Controls Dieser wichtige Knoten enthält die Komponenten, die sich innerhalb des Fensters befinden. Während Containerkomponenten dazu dienen, das

Layout des Fensters vorzugeben, dienen die Parts als tatsächlicher Inhalt. Placeholder bieten als eine Art von Parts ebenfalls Inhalt an.

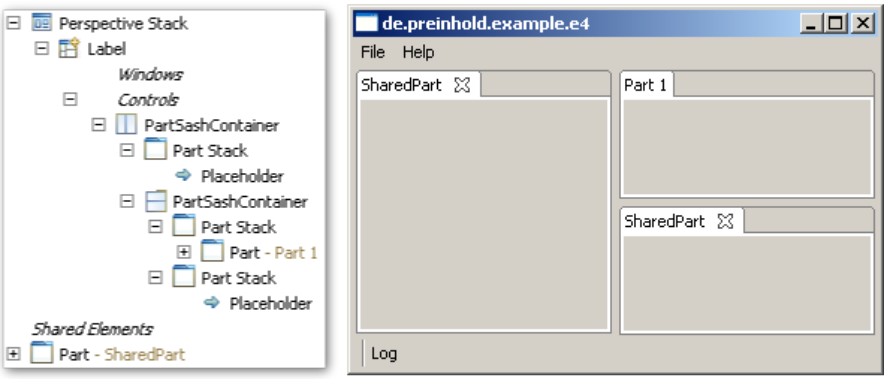

Abbildung 3.8: Layout und Screenshot einer e4-Applikation

Parts und Placeholder stellen die Blätter im Layoutbaum dar. Wie vom Bild abzuleiten, beziehen die Placeholder ihre Referenz aus den SharedElements der Window Komponente. Parts stellen einen essentiellen Bestandteil der e4-UI dar. Parts repräsentieren dabei Views und Editoren. Auf Abbildung 3.9 ist der obere Ausschnitt der Editorseite eines Parts im Workbench Modelleditor abgebildet.

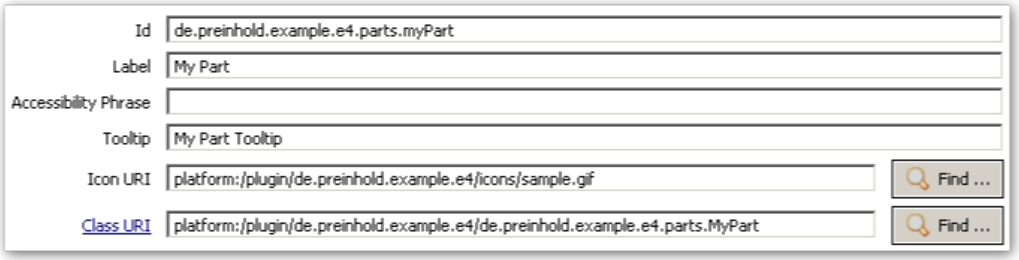

Abbildung 3.9: Editorseite eines Parts

Wichtige Felder für die grundlegende Verwendung sind das *Id* Feld und das *Class URI* Feld. Ersteres ist nötig um die Part-Komponente mit Hilfe des Part-Services identifizieren zu können. Die Class URI verknüpft analog zu Handlern die erstellte Java POJO mit der Part-Komponente im Modell. Eine beispielhafte Implementierung eines solchen POJO befindet sich im Anhang A.6

3.4.5 Services in e4-Applikationen

Das e4-Framework, speziell die IEC, bietet Zugang zu vorhandenen Eclipse Application Services, welche auch als *The 20 Things* bezeichnet werden. Diese Services beinhalten Komponeten wie Logging, Status Handling und Reporter, Modellmanipulationen sowie Event Handler Unterstützung [E4EAS]. Die Liste mit den entsprechenden konkreten Services befindet sich im Anhang A.4. Tatsächlich handelt es sich um weit weniger als 20 vorhandene Service, einige wichtige seien kurz vorgestellt.

EModellService Dieser Service befindet sich im *Application Context* und gewährt die Möglichkeit im Modell der Applikation zu suchen und dies zu bearbeiten. Das Modell kann dabei ebenfalls mittels @Inject Annotation injiziert werden.

```
1  @Inject
2  private EModelService service;
3
4  @Inject
5  private MApplication application;
6
7  // nach konkreten Objekten mitt els Ids suchen
8  service.findElements(application, "MyPart", MPart.class, null);
9
10 // nach Objekten eines Typs suchen
11 service.findElements(application, null, MPart.class, null);
12
13 // nach Objekten mit bestimmten Tags suchen
14 List<String> tags = new ArrayList<String>();
15 tags.add("tagname");
16 service.findElements(application, null, null, tags);
```

Listing 3.9: Implementierung EModelService

ESelectionService Der ESelectionService wird aus dem *Top Level Window Context* injiziert. Er erlaubt es auf die aktuelle Selektion der Workbench zuzugreifen. Dazu muss dem Service, neben dem Hinzufügen eines Listeners auch die Selektion bekannt gemacht werden. Anschließend kann mittels @Named-Parameter und Service Konstante auf diese zugegriffen werden.

```
1  @Inject
2  private ESelectionService service;
3  // component Listener hinzufuegen
4  component.addSelectionChangedListener(new ISelectionChangedListener() {
5    @Override
6    public void selectionChanged(SelectionChangedEvent event) {
7      IStructuredSelection selection =
8        (IStructuredSelection) component.getSelection();
9      service.setSelection(selection.getFirstElement());
```

```
10    }
11  });
12
13  //... Aufruf der Selektion an anderer Stelle
14  @Inject
15  public void setValue(@Optional
16              @Named(IServiceConstants.ACTIVE_SELECTION) Value val)
17  {
18    if(val != null) {
19      //mache etwas
20    }
21  }
```

Listing 3.10: Implementierung ESelectionService

EPartService Als Bestandteil des *Part Contexts* ist der Service im untersten Level der IEclipseContexts angesiedelt. Wie im vohergehenden Abschnitt angedeutet, ist es mit dem Service möglich, auf die Modellobjekte der Parts zuzugreifen, zu erzeugen und zu speichern. Letzteres führt die mit @Persist annotierte Methode des Part POJO aus.

```
1  @Inject
2  private EPartService service;
3  //...
4  MPart findpart = service.find("de.preinhold.example.e4.parts.myPart");
5
6  //Part per Descriptor erzeugen
7  service.createPart("de.preinhold.example.e4.myPartDescriptor");
8
9  //Part speichern
10 service.savePart(findPart);
11
12 //Part einblenden & aktivieren
13 service.showPart(findpart, PartState.ACTIVATE);
```

Listing 3.11: Implementierung EPartService

Eigene Services erzeugen und nutzen

Dieser Abschnitt soll anhand eines Beispiels sowohl die Erzeugung, als auch die Verwendung eines Services erläutern. Dabei wird im Beispiel ein OSGi Service erzeugt. Ein Service in OSGI ist durch eine Java Klasse oder ein Interface definiert. Übliche Praxis ist es den Service durch ein Plug-in zu definieren, welches ausschließlich das Modell und das Interface enthält. Dies erlaubt die Implementierung des Services in einem anderem Plug-in anzupassen.
Zunächst ist ein Plug-in für das Modell und das Serviceinterface zu erzeugen. Die Implementierung für diese befindet sich im Anhang A.8.
Das Plug-in, welches das Modell enthält, muss die Modellklassen und Interfaces im *Runtime* Tab per Export zur Verfügung stellen. Um das Modell mit

Daten zu füllen, soll an dieser Stelle eine Mock-Klasse erstellt werden, die den Service bereitstellt. Dazu ist ein neues Plug-in Projekt zu erstellen. Zu den Abhängigkeiten ist das eben erstellte Modellplug-in hinzuzufügen, zudem ist auf dem Übersichts-Tab der *MANIFEST.MF* Datei das Flag *Activate this plug-in when on of its classed id loaded*. Die Implementierung der Service-Klasse könnte wie folgt aussehen.

```
1  public class CarRetalMockModel implements ICarRentalModel {
2    private List<RentalOffice> rentalOffices;
3
4    public CarRentalMockModel() {
5      rentalOffices = new ArrayList<RentalOffice>();
6      RentalOffice oldClunkers = new RentalOffice();
7      oldClunkers.setRentalOfficeID(10001);
8      Car ford = new Car("Ford Explorer", 10001);
9      oldClunkers.addCar(ford);
10     ...
11     RentalOffice fancyCars = new RentalOffice();
12     fancyCars.setRentalOfficeID(10002);
13     Car bmw = new Car("BMW 760i", 20001);
14     fancyCars.addCar(bmw);
15     ...
16     rentalOffices.add(oldClunkers);
17     rentalOffices.add(fancyCars);
18   }
19   @Override
20   public List<RentalOffice> getRentalOffices() {
21     return rentalOffices;
22   }
23 }
```

Listing 3.12: Implementierung Service-Klasse

Um das Plug-in als OSGi Service verfügbar zu machen, wird zunächst ein Ordner mit dem Namen *OSGI-INF* im Plug-in Verzeichnis erzeugt. Im Anschluss wird im diesem Verzeichnis eine Komponenten Definition, mit folgendem Inhalt angelegt.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
3   name="de.preinhold.carRental.model">
4    <implementation
5     class="de.preinhold.carRental.model.servicemock.CarRentalMockModel"/>
6    <service>
7       <provide interface="de.preinhold.carRental.model.ICarRentalModel"/>
8    </service>
9 </scr:component>
```

Listing 3.13: XML Konfiguration OSGi Service

An dieser Stelle ist sicherzustellen, dass die Service Komponente in der Manifestdatei korrekt registriert wurde. Der Eintrag `Service-Component:OSGI-INF/component.xml` muss dazu in der MANIFEST.MF verzeichnet sein.

Um den Service zu nutzen, müssen die Depedencys zu den erstellten Plug-ins des Modells und des Mock-Services der e4-Applikation hinzugefügt werden. Abbildung 3.10

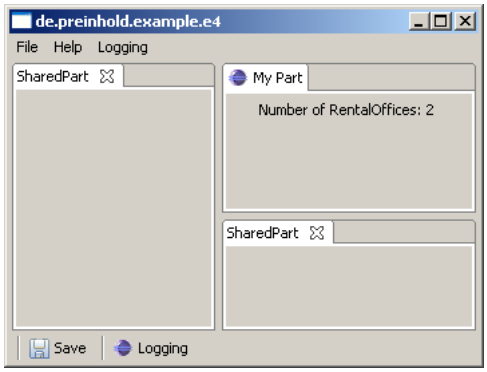

Abbildung 3.10: Screenshot der Programmoberfläche nach der Anpassung des MyPart POJO aus 3.8.

Am Ende des Abschnitts zusammenfassend die Erzeugung und die Nutzung eigener OSGi Services in einer e4-Applikation.

1. Plug-ins für Modell und Service-Interface erzeugen
2. Plug-in für die Service Implementierung erzeugen
3. OSGi-Komponenten Definition im Plugin der Service Implementierung hinzufügen
4. Dependencys hinzufügen, anschließend Service-Interface innerhalb der e4 Anwendung injizieren

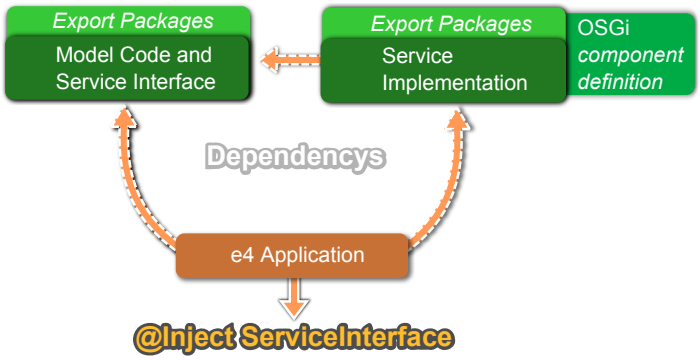

Abbildung 3.11: Prinzip der Implementierung eines OSGi Services

3.4.6 Contributions – Erweiterungen der e4-Applikation

Die E4AP erlaubt die Erweiterung des Applikationsmodells durch Modell-Elemente über andere Plug-ins. Der Extension Point *org.eclipse.e4.workbench.model* ist dafür der Ausgangspunkt. Erweiterungen können dabei durch sogenannte *fragments* oder *processors* vorgenommen werden [Vogel11a].

Fragments

Eine Möglichkeit der Erweiterung ist die der *Fragments*. Diese Erweiterungsart entspricht am ehesten, der unter 3.x typsichen Methode. Soll eine Contribution durch ein Fragment dem Applikationsmodell hinzugefügt werden, so muss das Element, an welchem die Contribution eingehangen wird, stets durch eine Id spezifiziert werden.
Modellfragmente können per Wizard erzeugt werden. Dazu ist im New Dialog unter e4 -> Model, *New Model Fragment* auszuwählen. Als Container ist das Plug-in zu wählen, in dem das Fragment erzeugt werden soll. Als Name kann der Standardname *fragment.e4xmi* gewählt werden.
Konkrete Fragmente werden mittels Selektion der Model Fragments im Fragment Editor und dem Add-Button hinzugefügt. Dabei handelt es sich um sogenannte String Model Fragmente. Die Element Id eines Fragments beschreibt den Einhängepunkt im Modell, welches erweitert werden soll. Durch Auswahl des Container-Typs, kann die Menge der zur Verfügung stehenden Einhängepunkte eingeschränkt werden. Wird als Typ *Application* gewählt, so können Applikationsmodelle, üblicherweise mit Application.e4xmi bezeichnet erweitert werden. Analog ist dies mit anderen Containern wie Menüs, Commands oder PartStacks möglich. Das zweite Feld im Fragmenteditor gibt die Art der Contribution an. Die üblichen Erweiterungen finden sich im Applikationsmodell unter Application.

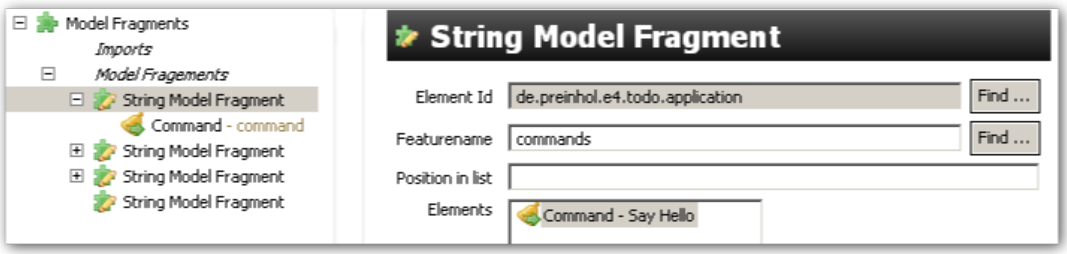

Abbildung 3.12: Fragment Editor. Ansicht eines String Model Fragments, welches das Applikationsmodell um einen Command erweitert.

Die Implementierung des Inhalts von Komponenten, wie Handler oder Parts, erfolgt wie im Applikationsmodell über annotierte POJOs. Ebenso werden die Modellkomponenten und POJOs mittels Class-URI verknüpft.

Abbildung 3.13: Fragment Editor. Ansicht eines String Model Fragments, welches das Applikationsmodell um einen Handler erweitert.

Um die Erweiterung zugänglich zu machen, wird im Plug-in, welches das Fragment beherbergt, dem Extension Point *org.eclipse.e4.workbench.model* ein *fragment* hinzugefügt. Dies geschieht im Tab Extensions in der plugin.xml. Als Property des Fragments ist der Pfad zum Modellfragment anzugeben.

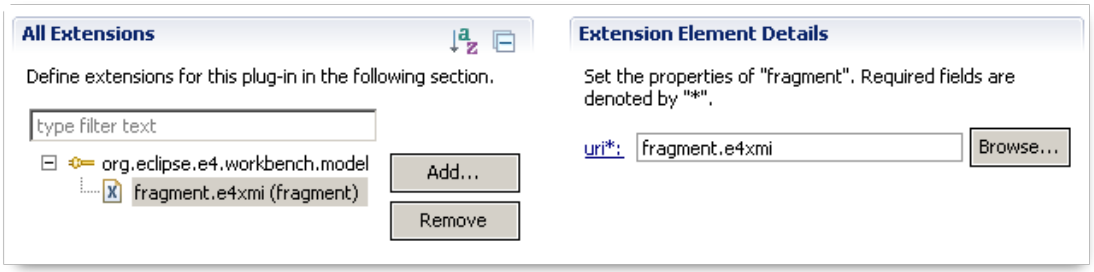

Abbildung 3.14: plugin.xml Editor. Ansicht der Extension.

Fragments eignen sich um statische Bestandteile der Applikation auszulagern und damit die Application.e4xmi des Hauptplug-ins übersichtlich zu halten. So könnten zum Beispiel umfangreiche Menüs oder Trimbars ausgelagert werden. Durch eine solche Aufteilung könnte eine Bearbeitung des Applikationsmodells ohne Komplikationen einfach auf mehrere Entwickler aufgeteilt werden.

Processors

Im Gegensatz zu Fragments ist es mit Processors neben dem Hinzufügen von Modellkomponenten auch möglich diese zu bearbeiten oder zu entfernen. Dadurch ist die Verwendung von Processors flexibler. Erreicht wird diese Flexibilität, indem das Modell per Java Code bearbeitet bzw. erweitert wird. Folgendes Beispiel verdeutlicht die Anwendung eines Processors zur Bearbeitung eines Menüeintrags zum Schließen der Workbench.

```
1  public class ExitDialog extends Dialog {
2    @Inject
3    public ExitDialog(@Named(IServiceConstants.ACTIVE_SHELL) Shell shell) {
4      super(shell);
5    }
6    @Override
7    protected Control createDialogArea(Composite parent) {
8      Label lab = new Label(parent, SWT.NONE);
9      lab.setText("Exit application ? Unsaved data will be lost.");
10     return parent;
11   }
12 }
```

Listing 3.14: Implementierung ExitDialog

```
1  public class ExitHandlerWithConfirm {
2    @Execute
3    public void execute(IWorkbench workbench, IEclipseContext context) {
4      ExitDialog dia=ContextInjectionFactory.make(ExitDialog.class, context);
5      dia.create();
6      if(dia.open() == Window.OK) {
7         workbench.close();
8      }
9    }
10 }
```

Listing 3.15: Implementierung erweiterter ExitHandler

Diese beiden Klassen stellen den Schließen Dialog und den dazugehörigen Handler dar. Sie gehören damit nicht unmittelbar zum Processor bzw. zur Erweiterung des Modells. In Zeile vier des `ExitHandlersWithConfirm` wird die typische Erzeugung eines Objekts mittels DI-Komponente aufgezeigt.
Die Processor-Klasse bedarf besonderer Aufmerksamkeit. In dieser wird über die Id des Einhängepunkts, die Modellkomponente injiziert. Die Id muss dabei mit jener, welche in der Extension festgelegt wird, übereinstimmen. Die Zeile 3 des Codeausschnitts auf der folgenden Seite verdeutlicht dies.
Das Modell des Einhängepunktes kann von da an beliebig verändert werden. In Listing 3.20 wird der Menüpunkt *Exit* gelöscht und dafür ein neuer Menüpunkt *Exit (with confirm)* hinzugefügt. Der Grund warum im `MenuItemProcessor` ein DirectMenuItem erzeugt wurde, liegt an der Tatsache, dass dieses MenuItem keinen Command benötigt.

```java
public class MenuItemProcessor {
  @Inject  @Named("de.preinhold.example.e4.fileMenu")  private MMenu menu;

  @Execute
  public void execute() {
    //alten Menueintrag entfernen
    if (menu != null && menu.getChildren() != null) {
      List<MMenuElement> list = new ArrayList<MMenuElement>();
      for (MMenuElement element : menu.getChildren()) {
        if (element.getLabel().contains("Exit")) {
          list.add(element);
        }
      }
    menu.getChildren().removeAll(list);
    }
    //neuen Menueintrag hinzufuegen
    MDirectMenuItem menuItem = MMenuFactory.INSTACE.createDirectMenuItem();
    menuItem.setLabel("Exit (with confirm)");
    menuItem.setContributionURI("platform:/plugin/<pluginName>/"+
    "<pluginName>.handler.ExitHandlerWithConfirm");
    menu.getChildren.add(menuItem);
  }
}
```

Listing 3.16: Implementierung MenuItemProcessor

Bei der Erweiterung mittels Processor muss zum einen die Processor-Klasse angegeben werden, und zum anderen die Id der Contribution, gewissermaßen des Einhängepunktes.

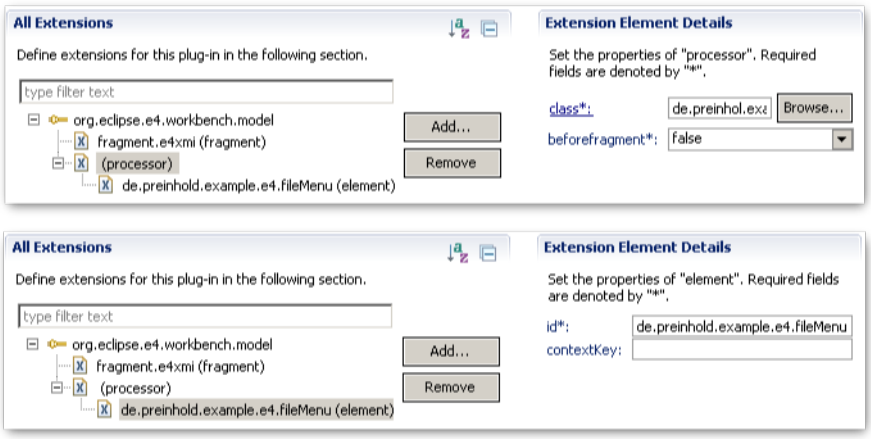

Abbildung 3.15: Contribution (Modellerweiterung) mittels Processor.

Mit Processors ist es möglich statische Teile des Applikationsmodells zu dynamisieren. So könnten abhängig von der Benutzergruppe einzelne Auswahlpunkte eines Menü angepasst oder entfernt werden. Wichtig ist das Flag *beforefragment* zu beachten, um dynamische Anpassungen nicht zu überschreiben.

3.4.7 CSS Styling von e4-Applikationen

Mittels Cascading Style Sheet Dateien, kurz CSS Dateien, lassen sich e4-Applikationen in ihrem Aussehen verändern. Dabei können Hintergrund, Schrift und Ränder von Komponenten der UI angepasst werden. Typischerweise besteht eine CSS Datei aus Regeln, basierend auf Selektoren und ihren Deklarationen. Deklarationen ihrerseits bestehen aus Eigenschaft und Wert. Folgender Codeausschnitt verdeutlicht die Verwendung der Sytlesheets.

```
1 .MTrimbar {
2   background-color: #cccccc;
3 }
```

Listing 3.17: CSS Einführungsbeispiel

Der Selektor ist in diesem Beispiel `.MTrimbar`, die Deklaration ist innerhalb der geschweiften Klammern, bestehend aus der Eigenschaft `background-color` mit dem Wert `#cccccc`.

Default und Themes in 4.x

Generell stehen zwei Möglichkeiten zur Verfügung, das Aussehen der Applikationen zu verändern. Eine erste Möglichkeit ist es eine *Default CSS Datei* anzugeben. Die Default CSS wird in der Produktdefinition angegeben und befindet sich üblicherweise in einem *CSS* Ordner im Plug-in Verzeichnis. Die Property in der Produktdefinition hat den Namen *applicationCSS* und als Wert den Pfad zur entsprechenden CSS Datei.

```
1 <property  name="applicationCSS"
2   value="platform:/plugin/de.preinhol.example.e4.csstest/css/default.css">
3 </property>
```

Listing 3.18: Konfiguration einer Standard-CSS in der plugin.xml

Diese Property wird beim Erzeugen einer e4-Applikation mit dem e4-Tooling automatisch erzeugt. Die Möglichkeit einer Default CSS Datei bietet sich besonders dann an, wenn nur ein Thema auf der UI verwendet werden soll. Soll dagegen eine Änderung des UI-Aussehens zur Laufzeit möglich sein, ist dazu größerer Aufwand von Nöten. Zum einem müssen mehrere CSS Dateien definiert werden, für jedes Thema eine Datei. Weiterhin muss `org.eclipse.e4.ui.css.swt.theme` zu den Dependencys hinzugefügt werden. Im Gegensatz zum Default CSS Fall wird die folgende Eigenschaft zur Definition hinzugefügt.

```
1 <property   name="cssTheme"
2   value="de.preinhol.example.e4.rentacar.standardTheme">
3 </property>
```

Listing 3.19: Konfiguration eines Standard-CSS-Themas in der plugin.xml

Diese legt fest, welches der Themen beim ersten Start der Applikation geladen werden soll. Die Änderung des Themas zur Laufzeit kann ebenfalls in die Deltas übernommen werden, so dass das geladene Thema beim Applikationsstart sich von diesem in der Produktdefinition unterscheiden kann. Neben dieser Property ist es nötig die Themen zu definieren. Dies geschieht mittels Extension `org.eclipse.e4.ui.css.swt.theme` in der folgenden Form.

```
<extension point="org.eclipse.e4.ui.css.swt.theme">
<theme   basestylesheeturi="css/default.css"
  id="de.preinhol.example.e4.rentacar.standardTheme"
  label="standard">
</theme>
```

Listing 3.20: Themendefiniton mittels Extensions

Soll das Thema zur Laufzeit geändert werden, kann dies über einen Handler geschehen.

```
public class ThemeBlackHandler {
  @Execute
  public void switchTheme(IThemeEngine engine){
    engine.setTheme("de.preinhol.example.e4.rentacar.blackTheme", true);
  }
}
```

Listing 3.21: Implementierung ThemeHandler

Mittel `setTheme()` wird das neue Thema gesetzt, dabei ist die in der Themendefinition angegebene Id zu übergeben. Der boolean Parameter legt fest ob die Änderung des Themas in die Deltas übernommen werden soll. Ein Beispiel für eine CSS Datei, sowie die Übersicht der zur Verfügung stehenden Selektoren finden sich im Anhang A.5.
Neben den vorhandenen Selektoren können UI Komponenten Ids zugeordet werden um diese individuell zu gestalten. Die Zuweisung zeigt folgender Codeausschnitt.

```
Label specialLabel = new Label(parent, SWT.NONE);
specialLabel.setText("Special Label");
specialLabel.setData("org.eclipse.e4.ui.css.id", "specialLabel");
```

Listing 3.22: Implementierung specialLabel

Die Anpassung des Aussehens kann in der CSS Datei in der folgenden Form geschehen.

```
#specialLabel {
  background-color: red #ffffff 50%;
  color: rgb(200,200,200);
  font: Verdana 12px;
}
```

Listing 3.23: CSS Spezial-Label

Ebenfalls sind im obigen Codeausschnitt die Möglichkeiten der Farbdefinition zu sehen. Entweder über vordefinierte Farbbegriffe, Hexadezimalwerte oder der rgb-Funktion. Im Falle der Hintergrundfarbe handelt es sich um einen linearen Farbverlauf oder Gradienten von rot zu weiß innerhalb der ersten 50% der Labelhöhe. Auch radiale Gradienten werden unterstützt. Eine Übersicht über mögliche Farbverläufe findet sich im Anhang A.5.

Styling unter 3.x

Für das Styling mittels CSS unter Eclipse 3.x ist eine Installation der e4-CSS Support Plug-ins notwendig [Toedt10b]. Die dazu notwendigen Bundles sind folgend aufgelistet.

- org.apache.batik.css
- org.apache.batik.util
- org.apache.batik.util.gui
- org.apache.batik.xml
- org.eclipse.e4.ui.css.core
- org.eclipse.e4.ui.css.jface
- org.eclipse.e4.ui.css.swt
- org.eclipse.e4.ui.css.swt.theme
- org.eclipse.e4.ui.widgets
- org.w3c.css.sac
- org.w3c.dom.smil
- org.w3c.dom.svg
- org.eclipse.equinox.ds

Innerhalb der Run Configuration ist unter dem Plug-in-Tab mit dem *Add Required Plug-ins* Button sicher zu stellen, dass alle benötigten Bundles geladen werden. Des Weiteren sind innerhalb des `org.eclipse.e4.ui.css.swt` Bundles alle Abhängigkeiten zu org.eclipse.e4.core zu entfernen. Um ein Thema nutzen zu können ist es, wie mit Eclipse 4.x, notwendig, das Thema über die Extension `org.eclipse.e4.ui.css.swt.theme` zu definieren. Die Verwendung des Themas wird in der createWorkbenchWindowAdvisor() Methode des `ApplicationWorkbenchAdvisor` angegeben.

```
1 ...
2 Bundle b=FrameworkUtil.getBundle(getClass());
3 BundleContext c=b.getBundleContext();
4 ServiceReference sRef=c.getServiceReference(IThemeManager.class.getName());
5 IThemeManager tManager=(IThemeManager) c.getService(serviceRef);
6 final IThemeEngine e=tManager.getEngineForDisplay(Display.getCurrent());
7 e.setTheme("de.preinhol.example.cssMail.standardTheme", true);
8 ...
```

Listing 3.24: Verwendung eines CSS-Themas unter Eclipse 3.x

3.4.8 Rendering von e4-Applikationen

Die Renderengine, ist lose mit dem Applikationsmodell gekoppelt [Schin11c] und [SchiBo10]. Dies bringt einige Besonderheiten mit sich. Die Illustration des Rendermodells soll als Ausgangsposition dienen. Im weiteren Verlauf des Abschnittes soll mit Hilfe eines konkreten Beispiels vertiefend auf die Entwicklung eines eigenen Renderers, eingegangen werden.

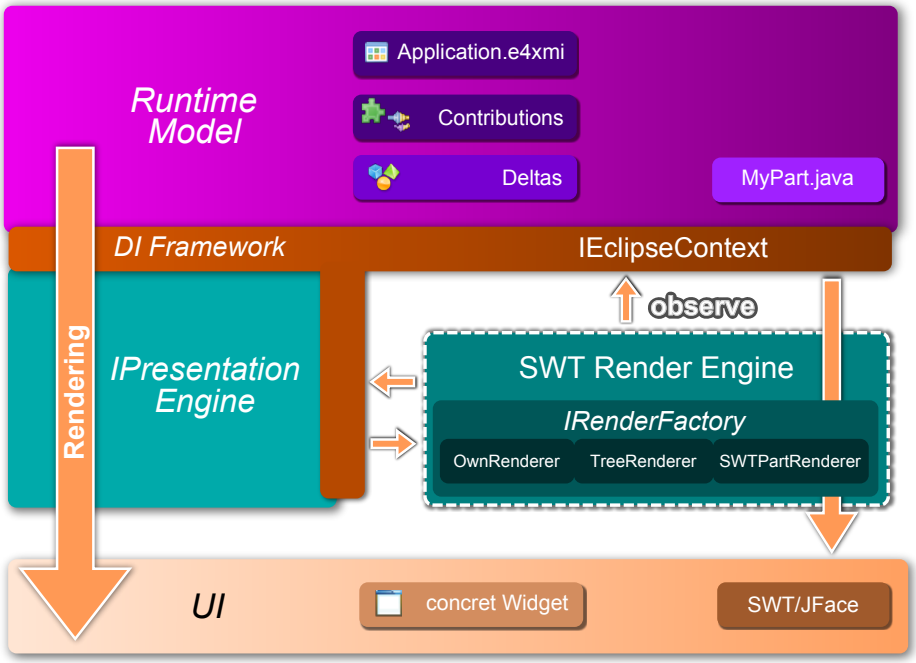

Abbildung 3.16: e4-Render Modell

Die Abbildung stellt den Standardfall des Rendermodells dar und kann sowohl von oben nach unten, wie auch von links nach rechts interpretiert werden. Die Farbverläufe von links nach rechts sollen eine Konkretisierung des Modells darstellen. Ausgehend von der linken oberen Ecke, wird das abstrakte Runtime-Modell der Applikation mithilfe einer IPresentationEngine gerendert, um auf der Benutzeroberfläche(UI) dargestellt zu werden. Konkreter bedeutet dies, das Laufzeitmodell der Workbench[13] gelangt mit Hilfe des DI Frameworks an die IPresentationEngine, welche den geeigneten Rederer für die Komponente des Modells bereitstellt, um so das konkrete Widget anzeigen zu können. Die rechte Seite des Modells spiegelt ein konkretes Beispiel wider. Ein Part-Modell MPart wird mit der SWT-Renderengine gerendert. Der Part ist mittels IEclispeContext verbunden, ebenso wie die SWT Engine. Innerhalb der Engine sucht eine

[13] Vgl. Abbildung 3.2

IRenderFactory nach den entsprechenden Renderern für das Part-Modell. Die Standard Render Factory ist die `WorkbenchRenderFactory`, welche mit einem großen *if/else Block* verglichen werden kann [Schind11d]. Diese wählt im Falle des Modellbeispiels den SWTPartRenderer aus, welcher das SWT-Composite als Widget erzeugt und auf der UI anzeigt.
Für die Anpassung der Renderengine gibt es zwei prinzipielle Möglichkeiten. Zum einen, wie mit Hilfe der Strichlinie in der Modellabbildung angedeutet, der Austausch der kompletten Engine. So kann die Standard SWT Engine beispielsweise durch eine Swing-, JavaFX-, QT-Engine ersetzt werden. Der folgende Abschnitt *Implementierung einer Renderengine* geht auf diesen Fall näher ein. Eine zweite Möglichkeit ist es die vorhandene Renderengine zu erweitern. Im Standard Fall, die SWT Engine. Der Abschnitt *Erweiterung einer Renderengine* geht mit einem Beispiel genauer auf diesen Fall ein.

Implementierung einer Renderengine

Für die Implementierung einer vollständigen Engine ist laut [Schind11d] das IPresentationEngine Interface der Ausgangspunkt. Die Schnittstelle besteht aus 5 Methoden.

```
1  public interface IPresentationEngine {
2   public Object createGui(MUIElement element, Object parentWidget,
3   IEclipseContext parentContext);
4
5   public Object createGui(MUIElement element);
6
7   public void removeGui(MUIElement element);
8
9   public Object run(MApplicationElement uiRoot, IEclipseContext appContext);
10
11  public void stop(); }
```

Listing 3.25: Interface IPresentationEngine

Die IPresentationEngine ist die zentrale Stelle in welcher die e4-Applikation der Aufbau der UI und die Synchronisation mit dem Applikationsmodell gesteuert wird. Die `createGui()` Methode ist dafür verantwortlich passende Renderer, parent-Komponenten und dazugehörige Context zusammenzuführen. Die `run()` Methode stößt als Start Methode die Erzeugung der UI-Widgets an, bietet aber auch die Möglichkeit beispielsweise das Look and Feel einer Swing e4-Applikation festzulegen. Weitere Bestandteile einer Renderengine ist die IRendererFactory und deren Implementierung, sowie die Implementierung der einzelnen Renderer für die Modellelemente.

Erweiterung einer Renderengine

Eine weitere Möglichkeit den Rendervorgang einer e4-Applikation zu beeinflussen, ist die Erweiterung der Renderengine. Als Beispiel zur Illustration soll das Hinzufügen einer *Open Street Map (OSM)*[14], Komponente dienen. Dieses orientiert sich am Tutorial von [Vogel10]. Dazu werden in diesem Absatz die folgenden Schritte beschrieben.

1. Die Erweiterung des bestehenden Applikationsmodells mit dem Eclipse Modeling Framework.

2. Die Erzeugung einer RenderFactory und eines Renderers.

3. Die Erzeugung der Komponentenbestandteile (Java Script, HTML, CSS).

Die Erweiterung des Applikationsmodells mittels EMF ist der erste Schritt um die Renderengine zu erweitern. Voraussetzung dazu ist das EMF SDK Plug-in. Einem neu angelegten Plug-in Projekt sind die Dependencys org.eclipse.e4.ui.model.workbench, org.eclipse.e4.core.contexts und org.eclipse.e4.core.di hinzuzufügen. Im erstellten Ordner *model* wird ein neues Ecore Modell angelegt. Das EPackage wird mit dem Namen `extensions`, der Ns Prefix mit `de.preinhol.example.e4.renderer.modelextension.model` und der NS Uri mit dem Wert `http://www.preinhol.de/ui/e4/extensions` versehen. Anschließend wird mittels Rechtsklick *Load Resource ...* die Ressource geladen, welche das angelegte Modell erweitern soll. Mit der Auswahl 'Browse Registered Packages' wird eine Auswahl der vorhandenen Modelle angezeigt. Nach der Eingabe $*$ UI ist das erste Modell auszuwählen. Dieses entspricht dem e4-Applikationsmodell. Die untere Abbildung verdeutlicht dies.

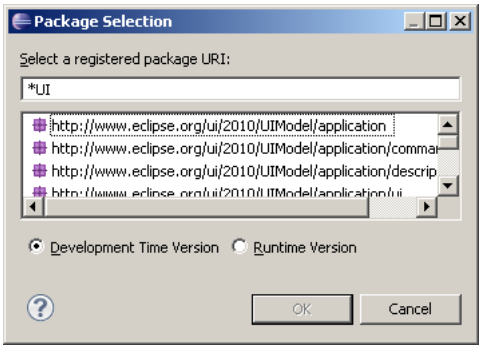

Abbildung 3.17: EMF Package Selection

[14] `http://www.openstreetmap.org/`

Im EPackage extensions wird nun die neue Komponente und der Container angelegt.

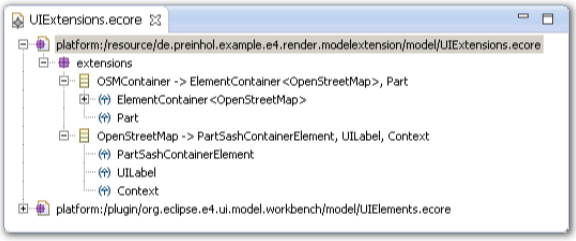

Abbildung 3.18: Erweiterung

An dieser Stelle sei darauf hingewiesen, dass das Betrachten des Modells mit dem Sample Reflective Ecore Model Editor auch die Referenzen der Super Typen anzeigt.[15] Dieser Editor ist dem Sample Ecore Model Editor vorzuziehen, da anderenfalls nicht der korrekte EClassifier(OpenStreetMap) der Containerkomponente angegeben werden kann. Auf Grundlage des Ecore Modells ist im Modellordner ein Genmodell zu erzeugen und mit diesem wiederum der Java Modellcode zu generieren. Die erzeugten Pakete sind im Runtime Tab der plugin.xml den zu exportierenden Paketen hinzuzufügen. Zudem ist das Anlegen eines Extension Point nötig.
Die Extension `org.eclipse.emf.ecore.generated_package` definiert sich aus einer wählbaren URI, der Paketklasse und dem Genmodell. Die URI hat die Aufgabe das neue Modell in der Application.e4xmi der e4-Applikation bereit zu stellen.

Die Erzeugung der RenderFactory und des Renderers wird in einem weiteren Plug-in vorgenommen. Dazu wird eine leere e4-Applikation generiert. Im src Ordner wird ein Package *renderer* angelegt in welchem die Klassen `MyMapRenderer` und `MyRenderFactory` erzeugt werden. Der Renderer und die Factory Implementierung ist im Anhang A.9 zu finden.
Der gerenderte Inhalt des Parts besteht aus einer SWT-Composite mit einer Browserkomponente, welche eine HTML Datei anzeigt und Java Script Befehle ausführt. Die Factory des Renderers besteht, genau wie die WorkbenchRenderFactory, aus if-else Blöcken, welche den korrekten Renderer für die UIElemente bereitstellt. Diese wird als Property im Extension Point der Produktdefinition mit dem Namen *rendererFactoryUri* und dem Pfad der Factory in der Form *platform:/plugin/*, mit anschließenden Plug-in- und vollständigem Klassennamen festgelegt. Neben der Erzeugung der Property muss die Dependency zum

[15] Rechtsklick, Open With Sample Reflective Ecore Model Editor

Modell-Erweiterungsplug-in hinzugefügt werden. Auf die neue Komponente kann nun wie folgt in der Application.e4xmi innerhalb der e4-Applikation zugegriffen werden.

```
1 <children xsi:type="extension:OSMContainer" xmi:id="..."
2  elementId="partID" label="Open Street Map">
3   <children xsi:type="extension:OpenStreetMap" label="my OSM">
```

Listing 3.26: Implementierung MenuItemProcessor

Die Voraussetzung für eine derartige Verwendung der Open Steet Map Komponente ist die Definition des Namespace im application-Tag.

```
1 xmlns:extension="http://de/preinhol/e4/render/model/UIExtensions.ecore">
```

Listing 3.27: Namespace-Erweiterung im application-Tag

Jedoch bringt diese Art der Namespace-Erweiterung das Problem mit sich, dass die Application.e4xmi nicht mehr mit dem Workbench Modelleditor geöffnet werden kann. Die Fehlermeldung gibt als Grund ein unbekanntes Paket mit der im Tag stehenden URI an.[16]

Die Erzeugung der Komponentenbestandteile in Form von HTML, CSS und JavaScript Dateien ist der abschließende Schritt zur Erweiterung der Renderengine. Dazu wird im Beispielplug-in ein `html` Ordner angelegt, in dem sich die im Renderer verwendete *map.html* befindet. Im Falle der Open Street Map Komponente werden die benötigten CSS Dateien und die benötigten JavaScript Dateien in Unterordnern angelegt.

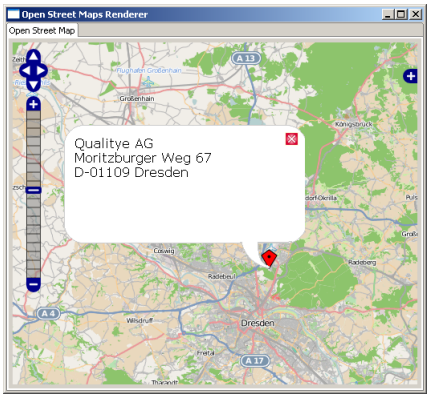

Abbildung 3.19: Screenshot einer um eine Open Street Map erweiterterte Renderengine.

[16] org.eclipse.emf.ecore.xmi.PackageNotFoundException: Package with uri 'http://de/preinhol/example/e4/render/model/UIExtensions.ecore' not found.

3.4.9 e4 und Eclipse RAP

Die Eclipse Rich Ajax Platform (RAP) bietet die Möglichkeit Eclipse RCP-Anwendungen innerhalb eines Browsers auszuführen. Abgesehen von der Laufzeitumgebung auf einem Server unterscheidet sich die RAP von der RCP im wesentlichen durch die Mehrbenutzerfähigkeit. Eine RCP-Applikation läuft lokal, damit hat jeder Benutzer eine Instanz des Programms zugewiesen, wogegen sich die Benutzer der RAP-Anwendung diese Instanz teilen. Gelöst wird dieses Problem durch die Einführung von Sessions und damit benutzerspezifischen Anwenderdaten. Die Entwicklung einer Anwendung für beide Plattformen wird als *Single Sourcing* bezeichnet.

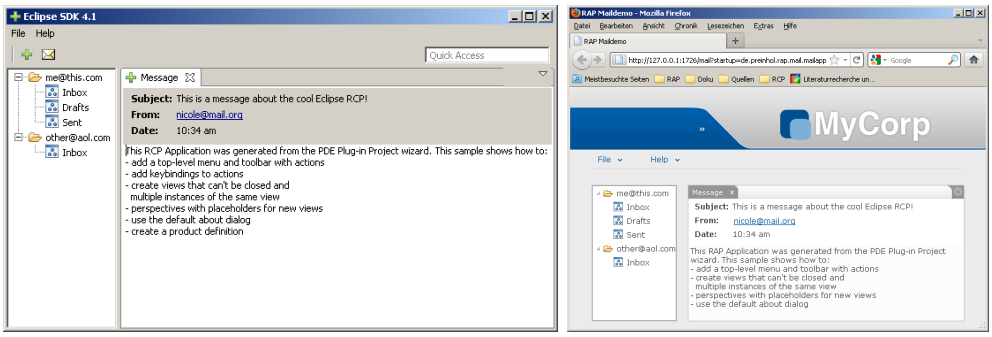

Abbildung 3.20: Screenshots der 3.x RCP-, sowie der RAP-Mail-Applikation.

Durch das zentrale Workbench Modell liegt die Annahme nahe, dass die Portierung dieses Modells für die Rich Ajax Plattform ohne größeren Aufwand möglich ist. Dies bestätigt die Aussage von [Sternb11], Co-Project-Lead der RAP, in einem Interview *Viele Konzepte [...] sind für RAP sehr interessant, da sie die Einschränkungen der 3.x-Plattform aufheben.* Weiterhin bestätigt er, dass das RAP-Entwicklungsteam sich anfangs rege am e4-Projekt beteiligt und gezeigt hat, dass RAP und e4 zusammen funktionieren. Beispiel-Applikationen aus dem Jahr 2009 belegen dies. [E4RAP] verweist jedoch darauf, dass die Demo-Applikationen höchst experimentell sind. So es mit dem aktuellen 4.1 Release nicht mehr möglich, diese Applikation zu starten. Auf das fehlende Aktualisierungsinteresse geht Sternberg im weiteren Verlauf des Interviews ein *Da [...] aus der RAP-Community noch kein großes Interesse an e4 wahrzunehmen war, haben wir uns erstmal anderen Themen zugewandt.* Die Integrationsbeschreibung des [E4RapIn], die seit 2009, teils seit 2008 unverändert besteht, bestätigt diese Aussage. Durch mangelndes Interesse der RAP-Community und den Aussagen Sternbergs ist davon auszugehen, dass das folgenden RAP Update im September 2011 kaum Verbesserungen für den e4-Support bringen werden.

3.5 Evaluierung des Eclipse 4.x Frameworks

Die bisherige Analyse der Arbeit ging der Fragestellung *Was kann das Framework?* nach . Zur Bewertung der Produktionstauglichkeit ist per Definition in 1.2. zudem die Analyse der Fragestellung *Was wird benötigt?* erforderlich.
Hier ist das Hauptproblem, da durch mangelnde Kenntnis des bisheriges Entwicklungsprozesses, dieser in der Qualitype AG nicht beurteilt werden kann. Und damit keine Aussage getroffen werden kann was benötigt wird. Die Lösung ist ein Dialog mit den Entwicklern der Qualitype AG. Um die entsprechenden Informationen zu erlangen, wird die Diskussion einen entsprechenden Aufbau haben.

1. Wieso wird mit dem Eclipse 3.x Framework entwickelt?

2. Welche Probleme gibt es bei der Entwicklung mit dem aktuellen Framework?

3. Welche Verbesserungen werden von einem neuen Framework erwartet?

4. Wie werden die folgenden Neuerungen bewertet?

   - zentrales und flexibles Applikationsmodell
   - wegfallende Editor View Unterscheidung mit durch POJOs geliefertem Inhalt
   - Dependency Injection
   - optionale Perspektiven
   - Modellerweiterung mittels EMF
   - CSS Styling

5. Laut [Sternb11] wird 4.2 (Juno) primäre Eclipse Version, wie sinnvoll ist es schon auf 4.1 umzusteigen?

Anhand der Antworten der Entwickler sollen Bewertungskriterien gefunden werden, um die Produktionstauglichkeit einschätzen zu können. Die Reihenfolge der Abschnitte entspricht der Nummerierung der aufgelisteten Fragen. Die Diskussion erfolgte mit dem Entwicklungschef und rund 10 Entwicklern der Qualitype AG.

Wieso wird mit dem Eclipse 3.x Framework entwickelt? Bevor mit dem Eclipse 3.x Framework entwickelt wurde, wurde mit Hilfe eines kleinen eigenen Frameworks auf Java Swing Basis entwickelt, welches auch immer noch genutzt wird (QualiTrail). Auf das Eclipse RCP-Framework wurde mit steigenden Anforderungen der zu entwickelnden Applikationen auf Grund nicht vorhandener Alternativen gewechselt. Entscheidend dabei war eine modulare und erweiterbare Plattform, die native Widgets unterstützt. Nach jahrelanger Entwicklung stellten sich vor allem die Plug-in Gliederung, das standardisierte Dialog- und Wizardframework und die Applikationsplattform des Frameworks als besonders nützlich heraus. Features wie Auto-Compilation, das Command-Handler-Prinzip und die relativ leicht erstellbare Online Hilfe finden Zuspruch bei der Entwicklung.

Welche Probleme gibt es bei der Entwicklung mit dem aktuellen Framework? Trotz der standardisierten UI Bibliothek *SWT* ist bei der Implementierung der GUI ein hoher Aufwand von Nöten. Zudem ist die Plattformunabhängigkeit, etwa im Falle des Fokus' bzw. des aktiven Fensters, nicht vollständig gegeben, dadurch resultieren in einigen Fällen Probleme mit dem Databinding. Weiterer Kritikpunkt war die nicht benötigte Mächtigkeit des OSGi Frameworks, wie die mögliche Aktivierung bzw. Deaktivierung von Plugins zur Laufzeit der Applikation und der damit entstehende Overhead. In diesem Zusammenhang wurde von den Entwicklern die fehlende Transparenz im Bundlemanagement kritisiert. Von einigen Entwicklern wurde die Problematik des Dirty Flags angesprochen. Zum Beispiel ist die Übernahme der Daten erst nach dem expliziten Speichern manchen Endanwendern nicht selbsterklärend. Abschließend wurde sich zudem allgemein zur Entwicklungsumgebung geäußert. So behindern Bugs in der Eclipse IDE, im Clean- und Cachebereich zum Teil erheblich die Entwicklung

Welche Verbesserungen werden von einem neuen Framework erwartet? Wichtig war den Entwicklern die Stabilität der Features und dass diese wirklich wie versprochen umgesetzt sind. Im gleichem Atemzug wurde auch eine Unterstützung der neuen Features seitens der IDE erwartet. Mit Hilfe der Dependecy Injection sollen die unübersichtlichen Vererbungshierarchien wegfallen und damit die Entwicklung transparenter, sowie weniger Boilerplate-Code notwendig machen. In diesem Zusammenhang ist die Entkopplung der Komponenten sehr erwünscht. Weiterhin wird sich durch die neuen Gestaltungsmöglichkeiten ein verringerter IDE Charakter der GUI erhofft.

Wie werden die Neuerungen bewertet? Die Neuerungen werden durchweg positiv bewertet, vor allem von der Dependency Injection und den annotieren POJOs wird sich eine bessere Transparenz in der Entwicklung erhofft. Vom den CSS Styling wird sich zudem ein höherer Wiedererkennungswert der Software versprochen.

Laut [Sternb11] wird 4.2 (Juno) primäre Eclipse Version, wie sinnvoll ist es schon auf 4.1 umzusteigen? Trotz der positiven Erwartungen an das neue Frameworks, ist die Bereitschaft, auf dieses umzusteigen, verhalten. So wird nach Aussagen der Entwickler auch die zukünftige Eclipse Version 3.8 bevorzugt werden. Dies lässt sich damit begründet, dass umfangreiche Software mit dem 3.x Framework entwickelt wurde und der Aufwand, auf das neue Framework umzusteigen, nicht abgeschätzt werden kann. Von Verbesserungen im Bereich der Stabilität und Kompatibilität für den Entwicklungsprozess kann mit dem Eclipse 4.2 SDK ausgegangen werden. Inwiefern im Juni 2012 tatsächlich die Wahl ausfällt, ist daher rein spekulativ. Ein Umstieg auf das aktuelle Eclipse SDK 4.1 ist nicht vorgesehen.

Auswertung der Diskussion mit den Entwicklern der Qualitype AG

Die Diskussion wurde im Sinne der Evaluierung des Eclipse RCP-Frameworks 4.1 geführt. Das heißt, der Ablauf orientierte sich an den obigen Fragen, jedoch ohne explizit darauf zu verweisen, dass das Ziel ein Katalog an Kriterien zur Bewertung der Produktionstauglichkeit ist. Im Rahmen der Auswertung der Diskussion wurde dann mit Entwicklern Rücksprache gehalten, inwieweit einzelne Kriterien tatsächlich für die Beantwortung der Frage *Was wird benötigt?* relevant sind.

3.5.1 Kriterien zur Evaluierung

Anhand von Kriterien zur Evaluierung soll eine nützliche und objektive Beurteilung der Produktionstauglichkeit des Eclipse RCP-Frameworks 4.1 möglich sein. Als problematisch stellt sich dabei die Formulierung der Kriterien heraus. Allgemein formuliert würden diese zu leicht zu einem positiven Ergebnis führen, eine exakte Formulierung schafft eine starke Abgrenzung und damit unverhältnismäßig harte Kriterien. Die Gefahr ist, dass trotz gut gewählter Kriterien die Gesamtaussage zur Produktionstauglichkeit einem Münzwurf gleicht.
Die Kriterien orientieren sich weitgehend an den Aussagen der Entwickler. Da die Produktionstauglichkeit im Wesentlichen von der vollständigen Umsetzung der benötigten Funktionen abhängt und weniger von den Möglichkeiten, die das Framework bietet. Die folgend aufgelisteten Kriterien sollen als Bewertungsgrundlage für die Evaluierung der Produktionstauglichkeit dienen.

- modulare und erweiterbare Plattform
- lose Kopplung der Module
- native, performante Widgets
- standardisiertes Dialog- und Wizard-Framework
- Applikationsplattform, die den Rahmen der Anwendung sowie deren Laufzeitumgebung festlegt
- IDE Unterstützung neuer Framework Features
- stabile, fehlerfreie Funktion von angebotenen Features der IDE
- wenig Boilerplate-Code
- IDE Charakter der GUI der Applikation unerwünscht

3.5.2 Theorieorientierte Evaluierung

Die modulare, erweiterbare Plattform (Plug-in Gliederung) hat sich im Vergleich zur Version 3.x nur geringfügig geändert. Die grundlegende Gliederung in Plug-ins ist geblieben, da OSGi bzw. das Equinox Framework weiterhin als Fundament dient. Damit sind auch die Abhängigkeiten von und zu anderen Plug-ins geblieben. Neu ist das Prinzip wie der Inhalt der Plug-ins in die Gesamtapplikation eingefügt wird. So ist es über die beschriebenen Contributions möglich per Plug-ins das zentrale Applikationsmodell zu erweitern. Weiterhin müssen in diesen Zusammenhang auch Services genannt werden, welche genutzt werden können um Aspekte der Businesslogik in einem festgelegten Rahmen auszulagern.

Die lose Kopplung der Module wird mit Hilfe des neuen Programmiermodells wesentlich stärker unterstützt. So wird das Ziel der komponentenbasierten Softwareentwicklung durch die Dependency Injection noch effektiver gefördert. Auch in diesem Zusammenhang wird durch den serviceorientierten Ansatz des Frameworks die lose Kopplung bzw. die gezielte Nutzung der DI forciert.

An den Widgets hat sich im Vergleich zum 3.x Framework nichts geändert, da SWT nach wie vor die Standardbibliothek der grafischen Oberfläche ist. Durch das veränderte Rendermodell kann, mit zusätzlichem Aufwand, jedoch jede beliebige Bibliothek verwendet werden. Dies gestaltet die Verwendung nativer und performanter Widgets wesentlich flexibler. Laut [Schind11d] ist der Aufwand für eine Implementierung grundlegender Komponenten Renderer für die Java Swing Bibliothek in 5 Mannstunden durchaus akzeptabel.

Als standardisiertes Dialog- und Wizardframework verwendet die Version 4.1 des Frameworks vorerst weiterhin jenes der Version 3.x, daher ergeben sich hier keinerlei Änderungen.

Die Applikationsplattform, die den Rahmen der Anwendung sowie deren Laufzeitumgebung festlegt , bietet mit dem e4-Applikationsmodell eine stark verbesserte Möglichkeit an zentraler Stelle applikationsweite Änderungen vorzunehmen. Die E4AP basiert weiterhin auf der Equinox-Runtime, an dieser Stelle ergeben sich daher keine Unterschiede.

Als Unterstützung neuer Features , wie dem Applikationsmodell, bietet die Eclipse 4.1 SDK einen eigenen Editor. Dieser bietet einerseits eine übersichtliche Darstellung des Modells und andererseits eine Möglichkeit diese Modell einfach anzupassen. Eine Unterstützung des neuen Programmiermodells, also der Dependency Injection, ist dagegen weder vorhanden noch liegen Informationen vor, eine derartige Unterstützung bieten wollen.

Die stabile und fehlerfreie Funktion von angebotenen Features wie dem Workbench Modelleditor ist speziell nach der Erweiterung des Applikationsmodells durch neue Renderer und Komponenten nicht mehr nutzbar. In den anderen getesteten Fällen funktionierte der Editor einwandfrei. Installierte zusätzliche Plug-ins zur Entwicklung verursachten zum Teil Exceptions in der IDE und funktionierten erst nach einem Neustart der IDE fehlerfrei. Andere Plug-ins ließen sich durch bereits bestehende Plug-in Dependencys gar nicht installieren. Vor allem in den Meilensteinen M6 und M7 war dies besonders störend, mit Release der Version 4.1 verbesserte sich die Situation wesentlich.

Wesentlich weniger Boilerplate-Code ist durch das auf Annotationen basierende DI-Framework nötig. Lange und geschachtelte Aufrufe von getter Methoden sind daher weitgehend hinfällig geworden. Durch die Verwendung von POJOs fallen zudem lange Vererbungshierachien weg, die den Code unnötig aufblähen.

Der unerwünschte IDE Charakter der GUI wird durch das neue Partkonzept und die optionalen Perspektiven verringert. So hat eine e4-Applikation weit geringere Affinität zu einer IDE, als eine auf dem 3.x Framework basierende. Auch die Stylingmöglichkeiten mit Hilfe von CSS tragen dazu bei. Die Kombination dieser beiden Möglichkeiten liefert eine Benutzeroberfläche die an Eclipse erinnern mag, aber mit einer IDE nichts gemein haben muss.

Ausgehend von den Kriterien ist das Eclipse 4.1 RCP-Framework aus theoretischer Sichtweise für den Produktiveinsatz geeignet. Das Framework bietet wesentliche Verbesserungen in Bezug auf Modularisierung, lose Kopplung und Wiederverwertung, welche die fehlenden oder aktuell fehlerhafte Teile der IDE überwiegen. Inwiefern sich diese erste Evaluierung bestätigt, zeigt das folgende Kapitel.

4

Umstellung komplexer bestehender Anwendungen

Im Zuge der praktischen Analyse des Eclipse RCP-Frameworks 4.1 wird für eine bestehende Applikation ein Prototyp implementiert. Da die Umstellung einer komplexen Anwendung im Rahmen der Arbeit nicht möglich ist, wird anhand des implementierten Prototyps, der Portierungsaufwand der Applikation bestmöglich auf eine komplexe bestehende Anwendung übertragen. Eine solche Anwendung ist hierbei eine Java Enterprise-Applikation, deren Entwicklungsaufwand den eines gewöhnlichen Projektes ohne besondere Anforderungen in Umfang und Qualität übersteigt.

4.1 Umstellung einer konkreten Anwendung

Um den Aufwand für die Umstellung einer Applikation im Rahmen dieser Arbeit einschätzen zu können, soll ein firmeninterner Kontaktmanager auf die e4-RCP-Plattform portiert werden. Dieser Kontaktmanager dient zum Verwalten von Kundendaten, Kontaktprotokollen sowie zur Angebots-, Dokumenten- und Lizenzverwaltung. Im Zuge der Umstellung sollen auf bestehende Mängel, wie die fehlende Mehrbenutzerfähigkeit, Rücksicht genommen werden.

4.1.1 Vorbereitung

Zunächst muss geklärt werden, inwiefern die Umstellung erfolgen soll. Kann die bestehenden Anwendung mit angemessenem Aufwand portiert werden oder muss eine von Grund auf neue Applikation entwickelt werden?
Die bestehende Version des Kontaktmanagers ist eine Einzelnutzer-Anwendung auf Java Swing Basis. Der Zugriff auf diesen erfolgt über ein Netzlaufwerk. Die Nutzung der programminternen H2-Datenbank erfolgt exklusiv, so dass immer nur eine Instanz des Programms zur gleichen Zeit geöffnet sein kann. Die rund 300 bestehenden Kundendatensätze machen einen Datenmigration zwingend notwendig. Das Schema der Datenbank, sowie die Entitys sollen daher so weit wie möglich wiederverwendet werden. Ebenso soll das Design der grafischen Oberfläche weitestgehend übernommen werden.
Die Anforderungen an den neuen Kontaktmanager zielen weitestgehend auf die Forderung nach Mehrbenutzerfähigkeit ab. Eine Client-Server-Architektur soll dies ermöglichen. Das Frontend soll dabei auf den e4-Technologien aufbauen, für das Backend ist ein JBoss Application-Server der Version 6 mit angebundener H2-Datenbank vorgesehen. Die Anwendung ist zudem vollständig mit dem Eclipse 4.1 SDK zu entwickeln.
Da bis auf die Persistierung sämtliche Bestandteile des Programms überarbeitet werden müssen, stellt die Entwicklung des neuen Kontaktmanagers im Wesentlichen eine Neuentwicklung dar. Dies bietet den Vorteil, die e4-Technologien flexibel einzusetzen und damit keine Rücksicht auf vorhandene Strukturen nehmen zu müssen. Dieser Freiraum schafft jedoch den Aufwand alle Bestandteile der Applikation implementieren zu müssen. Weiterhin schlägt sich die mangelnde Kenntnis im Umgang mit den neuen Technologien in der Effektivität der Entwicklung nieder. Daher ist zu erwarten, dass der Prototyp nur grundlegende Aspekte, wie die CRUD-Operationen, des bestehenden Kontaktmanagers portieren kann.

Die in Abbildung 4.1 angestrebte Architektur orientiert sich an einer Client-Server-Architektur im Java Enterprise-Umfeld. Mit Hilfe von Enterprise JavaBeans (EJB) sollen Entitys und die serverseitige Businesslogik umgesetzt werden. Zudem sollen Entitys auch im Client als Behälter für Daten dienen, damit auf so genannte *ValueObjects* verzichtet werden kann.

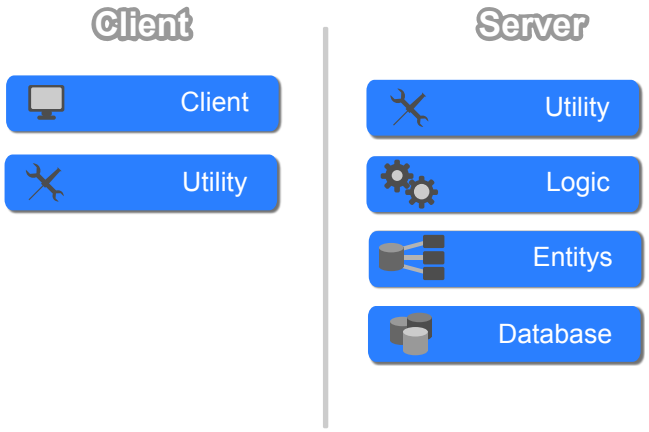

Abbildung 4.1: Eine vereinfachte Darstellung der Client-Server-Architektur des neuen Kontaktmanagers.

Zur Vorbereitung ist eine Testapplikation entwickelt worden, die sicherstellt, dass die e4-Technologien auch erfolgreich mit dem JBoss 6 zusammenarbeiten. In diesem Zusammenhang stellte sich eine Besonderheit im Umgang mit dem JBoss auf der Seite des Clients heraus. So benötigt das Plug-in, welches zur Kommunikation mit dem Server dient, eine *Buddy-Dependency*[1] zum JBoss Client Plug-in, da anderenfalls der Zugriff auf den JNDI-Kontext und damit auf die Persitenzschicht nicht möglich ist.
Die Daten werden im Client von einem OSGi-Service bereitgestellt, welcher analog zum Abschnitt 3.4.5 implementiert wurde. Damit ist die notwendige Buddy-Dependency ausschließlich auf das Implementierungsplug-in des Datenservice beschränkt. Mit Hilfe dieses Services können die Daten aus der Datenbank direkt in die Parts der e4-Applikation injiziert werden.
Neben der Gesamtarchitektur stellte sich in der Testapplikation die Aufteilung des e4-Clients in einzelne Plug-ins für UI-Komponenten, Services und dem Applikationsmodell als nützlich heraus. Orientiert wurde sich dabei an [Kutsc11]. Der folgende Abschnitt geht auf Architektur-, Datenfluss- und GUI-Implementierung ein.

[1] Normalerweise haben Plug-ins nur Zugriff auf die internen und importierten Klassen des abhängigen Plug-ins. Jedoch benötigt der JNDI-Kontext einen erweiterten Zugriff. Dazu dient das `Eclipse-BuddyPolicy` Flag in der Manifest Datei. Vgl. [IBM06]

4.1.2 Portierung

Die Abbildung 4.2 zeigt eine detaillierte Illustration der Gesamtarchitektur des neuen Kontaktmanagers. Die analoge Symbolik mit der Abbildung im vorherigen Abschnitt erleichtert das Verständnis.

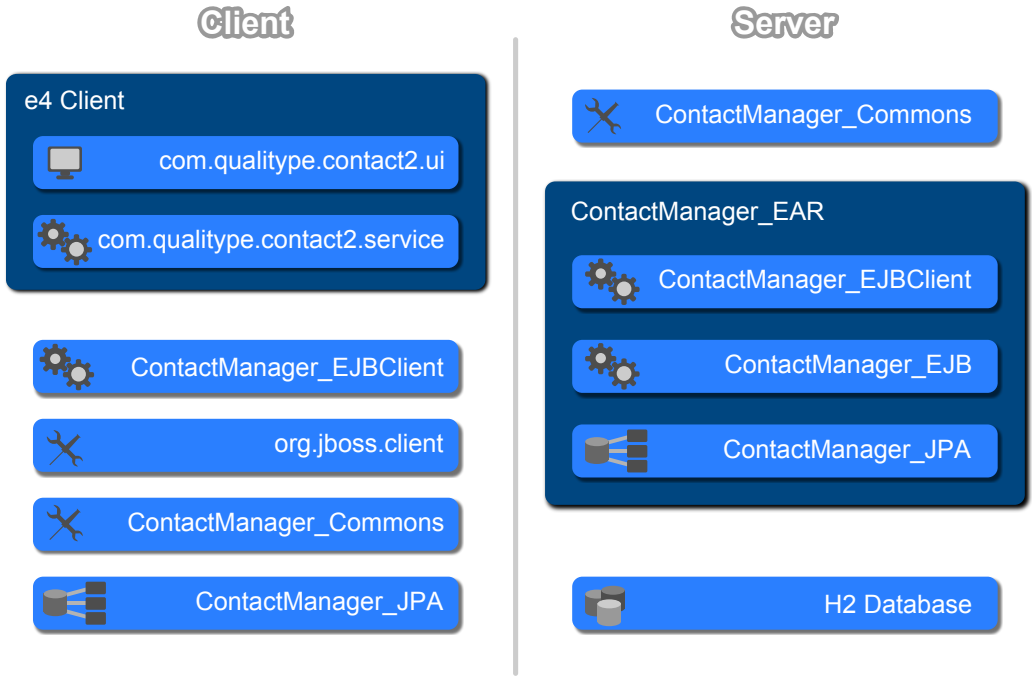

Abbildung 4.2: Vereinfachte Darstellung der konkreten Client-Server Architektur des neuen Kontaktmanagers.

Serverseitig besteht die Applikation aus dem `ContactManager_EAR` und der Commons Bibliothek, die auf Server und Clientseite notwendige Werkzeuge zur Verfügung stellt. Das Enterprise-Archive (EAR) wird dem JBoss zur Verfügung gestellt[2]und repräsentiert die vollständige serverseitige Anwendung. Es beinhaltet in Form von Jar-Archiven die Entity-Beans zum Mapping der Datenbank, die EJB SessionBeans zur Verwaltung der Zugriffe auf die Datenbank und die Schnittstellen für den Client, die den Zugriff auf die SessionBeans erlauben. Letztere müssen dem Client ebenfalls zur Verfügung stehen, um mit dem Server interagieren zu können. Da im Client auch die Entitys direkt verwendet werden, sind auch diese auf der Seite des Clients vorhanden. Die angedeutete JBoss Client-Bibliothek für die Buddy-Dependency befindet sich ebenfalls auf der Clientseite. Im Folgenden wird auf den für die Evaluierung relevanten e4-Client näher eingegangen.

[2]Als Deployment bezeichnet.

Analog zur Architekturdarstellung zeigt die Abbildung 4.3 den Aufbau des neuen Kontaktmanagers. Das Plug-in `com.qualitype.contact2` stellt mit dem e4-Applikationsmodell und der Produktdefinition das *Haupt Plug-in* dar.
Das `.service` Plug-in beinhaltet die Schnittstellen, die `.service.*` Plug-ins die Implementierungen der entsprechenden Services. Komponenten wie Dialoge, Parts, Handler und Wizards befinden sich im `.ui` Plug-in.

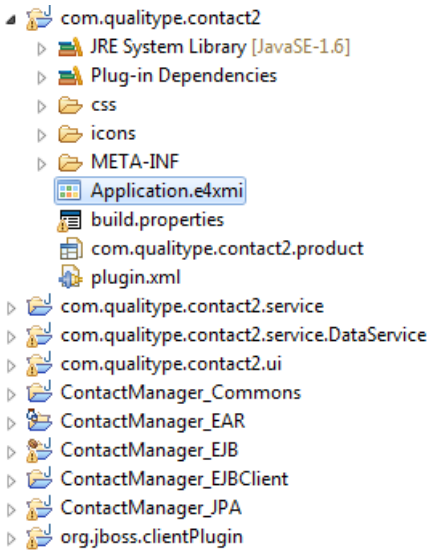

Abbildung 4.3: Aufbau des neuen Kontaktmanagers

Das EAR der Serverimplementierung sowie dessen Projekte und das JBoss-Client Plug-in sind neben den Client Plug-ins zu erkennen Die Bibliotheken des Servers werden im *lib* Ordner und Classpath des `.service` Plug-ins dem Client zur Verfügung gestellt. Da das UI Plug-in die Service-Interfaces benötigt, erlangt es dadurch ebenso Zugriff auf die Serverbibliotheken, die es bei direkter Verwendung der Entitys benötigt. Die derartige Strukturierung setzt die Architekturvorstellung aus Abbildung 4.2 exakt um. Die zusätzlich benötigten Dependencys zwischen den Client Plug-ins führen zu keinerlei Nachteilen, so ist der Zugriff auf andere Plug-ins etwa bei der Auswahl der Class-URI im Editor des Applikationsmodells problemlos möglich. Voraussetzung dafür sind entsprechende Dependencys in der *plugin.xml* des Haupt Plug-ins.
Um das DI-Prinzip des Programmiermodell effektiv zu nutzen, werden benötigte Daten mit Hilfe eines OSGi-Services injiziert. Konkret bedeutet dies: Der benötigte Inhalt eines Parts im Applikationsmodells wird durch ein *Part POJO* geliefert. In diesem POJO wird per *@Inject* der Service injiziert, der die benötigten Daten liefern soll. Diese Daten können mittels Widgets auf der grafischen Oberfläche angezeigt werden.

Die Abbildung 4.4 veranschaulicht den im Folgenden beschrieben Fluss der Daten im neuen Kontaktmanager.

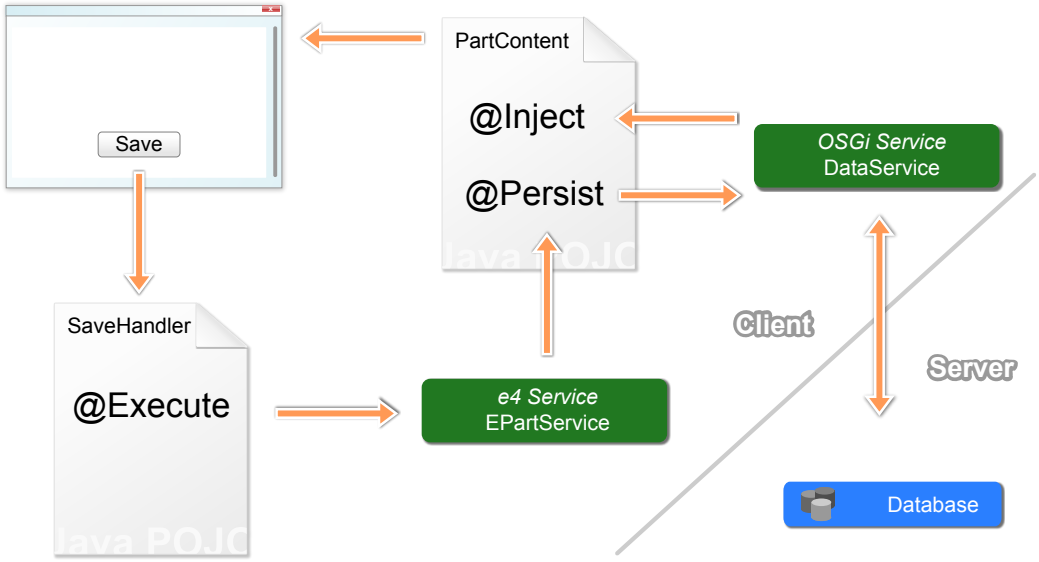

Abbildung 4.4: Veranschaulichung des Datenflusses innerhalb des e4-Clients

Das Listing 4.1 verdeutlicht wie eine UserEntity mit passender Id vom Service aus der Datenbank geliefert wird.

```
1 @Inject
2 public MyPart(DataService dataService) {
3   ...
4   userEntity = (UserEntity) dataService.getEntity(userEntity, userID);
5   ...
6 }
```

Listing 4.1: Vereinfachtes Beispiel DataService Kontaktmanager

Der Service seinerseits greift über einem vom JBoss bereitgestellten JNDI-Kontext auf entsprechende SessionBean Schnittstellen zu. Diese dienen serverseitig zur Verwaltung der Zugriffe auf die Datenbank. Auf diese Umsetzung wird hier nicht weiter eingegangen.

Um auf Änderungen der Daten durch den Nutzer auf der GUI zu reagieren, wird im `PartContent` mittels Databinding, einem e4-spezifischen Dirtymechanismus und einem `SaveHandler` das Speichern der veränderten Daten ermöglicht. Dazu wird im Part POJO ein *MDirtyable* Objekt injiziert. In diesem wird mit Hilfe des Databinding ein Dirty-Flag gesetzt. Das Listing 4.2 zeigt eine mögliche Implementierung des Databindings und das Setzen des Dirty-Flags.

```
@Inject private MDirtyable dirtyable;

@Inject
public MyPart(DataService dataService) {
  ...
  DataBindingContext bindingContext = new DataBindingContext();
  IObservableValue myModel =
    PojoProperties.value(UserEntity.class, "name").observe(userEntity);
  IObservableValue myWidget=
    WidgetProperties.text(SWT.Modify).observe(userName);
  myModel.addValueChangeListener(new IValueChangeListener() {
    @Override
    public void handleValueChange(ValueChangeEvent event) {
      dirtyable.setDirty(true);
    }
  });
  bindingContext.bindValue(myModel, myWidget);
  ...
}
```

Listing 4.2: Vereinfachtes Beispiel des Databinding-Dirtymechanismus

In diesem Listing wird die POJO-Property `name` mit einem Textfeld `userName` verknüpft. Durch einen Listener am Modell wird bei einer Änderung das Dirty-Flag gesetzt. Ziel des Dirty-Flags ist die Markierung eines Parts mit ungesicherten Werten. Durch einen simplen Handler kann dieser Dirtymechanismus ausgewertet werden. Das Listing 4.3 illustriert den vollständig zu implementierenden Handler, der nötig ist, um bei einem aktiven Part mit gesetztem Dirty-Flag das Speichern auszulösen.

```
public class SaveHandler {
  @Execute
  public void execute(@Active MPart part, EPartService partService) {
    partService.savePart(part, false);
  }

  @CanExecute
  public boolean canExecute(@Optional @Named(IServiceConstants.ACTIVE_PART)
  MDirtyable dirtyable) {
    if(dirtyable != null)
      return dirtyable.isDirty();
    else
      return false;
  }
}
```

Listing 4.3: Vollständige Implementierung eines SaveHandlers

Zudem zeigt das Codelisting die beiden Möglichkeiten auf den aktiven Part zu injizieren. Die @Named-Annotation könnte also genau so durch @Active ersetzt werden. Die @Optional-Annotation ist für den Fall nötig, dass kein aktiver Part existiert, etwa nach dem Programmstart. Innerhalb der execute()-Methode wird dem injiziertem PartService der zu speichernde Part übergeben

und damit die mit @Persist annotierte Methode des Part-POJOs aufgerufen. In dieser kann die vom DataService injizierte und vom Nutzer modifizierte Entity einem Service zum Speichern übergeben werden. Der Einfachheit halber soll der Datenservice auch das Speichern übernehmen. Listing 4.4 verdeutlicht das Vorgehen.

```
1  @Inject
2  public MyPart(DataService dataService) {
3    ...
4  }
5  @Persist
6  private void save(@Active MPart activePart, @Active Shell activeShell,
7     DataService dataService){
8
9    if(dataService.saveEntity(userEntity)) {
10     dirtyable.setDirty(false);
11     activePart.setLabel(userEntity.getLastName()+", "
12       +userEntity.getFirstName());
13     activePart.setIconURI("platform:/plugin/com.qualitype.contact2/"+
14       "icons/customer.png");
15   }
16   else {
17     MessageDialog.openError(activeShell, "Fehler", "Fehlerbeschreibung");
18   }
19 }
```

Listing 4.4: Vereinfachtes Beispiel einer Persist-Methode

Neben dem Speichern wird hier exemplarisch das Label und das Icon des gespeicherten Parts geändert und eine simple Fehlerbehandlung veranschaulicht. Bei erfolgreichem Speichern übergibt der Datenservice die Entity dem Server, welcher diese in der Datenbank ablegt bzw. aktualisiert. Die hier beschriebene Abhandlung folgt dem in Abbildung 4.4 illustrierten Ablauf, ausgehend von DataService, welcher die Daten injziert, bis zum Speichern der Daten, ebenfalls durch den Datenservice. Die zusammenhängende Implementierung des Part-POJOs findet sich im Anhang A.6

Die Bibliothek, welche die Grundlage für die grafischen Oberfläche liefert, wurde von Java-Swing auf SWT umgestellt. SWT ist der Standard von e4-Anwendungen und nutzt die Standard Widgets des Betriebssystems. Insofern möglich wurden jedoch die grafischen Komponenten der bestehenden Version eins zu eins übertragen oder verbessert. Das Childframe-Konzept wurde nicht übernommen, da es laut Anwender keine Verwendung gefunden hat. Es wurde gegen die Möglichkeit ausgetauscht, Kontakte in Karteireitern zu organisieren. Das bedeutet, das mehrere Kontakt gleichzeitig geöffnet sein können, zum Beispiel um Daten zu vergleichen. Zur besseren Bedienung sind die Reiter daher verhältnismäßig groß. Weitere Verbesserungen im Suchreiter betreffen die

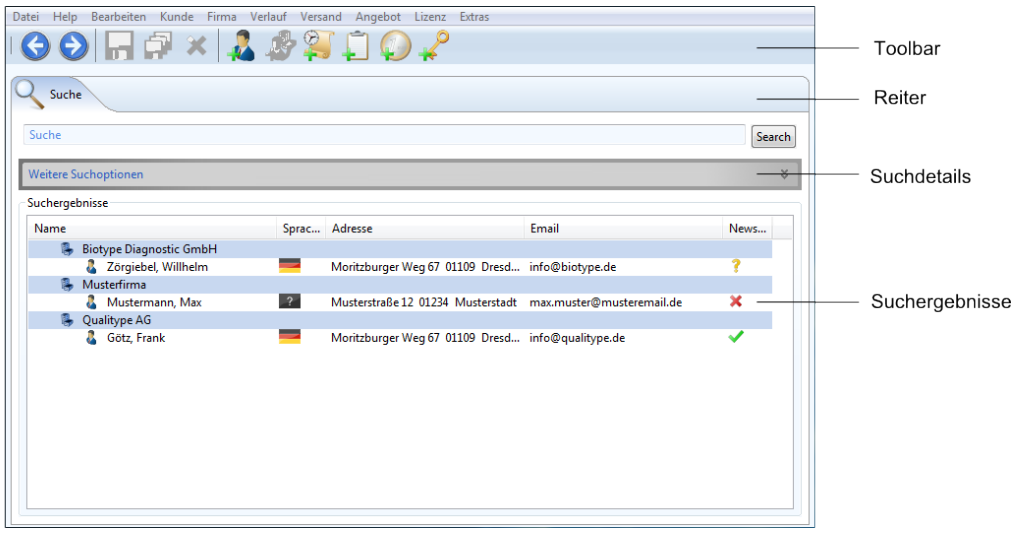

Abbildung 4.5: Beschrifteter Screenshot des neuen Kontaktmanager.

Suchdetails und die Anzeige der Suchergebnisse. Die Details sind per Expansion Bar ein- und ausblendbar um bei Bedarf mehr Platz für die Ergebnisse zu schaffen. Die Ergebnisanzeige wurde von redundanten Informationen befreit. Letzte wesentliche Änderung betrifft den Verlaufsreiter innerhalb eines Kontaktes. Die unübersichtliche Toolbar wurde gegen einzelne, den Eingabedaten zugeordnete Button ausgetauscht. Screenshots zum Vergleich beider Versionen finden sich im Anhang A.10.

Das e4-Applikationsmodell des neuen Kontaktmanagers ist sehr simpel. Es besteht aus einem Trimmed Window mit oben ausgerichteter Toolbar und einem PartStack für die Anzeige der Parts, auf Perspektiven wird demnach verzichtet. Die Parts entsprechen dabei den einzelnen Reitern. In obiger Abbildung 4.5 ist der Suchpart dargestellt. Dieser ist im Modell als nicht schließbar markiert und erlangt seinen Inhalt von dem POJO `SearchPart` aus dem Plug-in `com.qualitype.contact2.ui`. Jedem Icon der Toolbar wurde im Applikationsmodell ein Command und ein Tastaturkürzel zugewiesen. Alle weiteren Inhalte werden dynamisch zur Laufzeit erzeugt. Erwähnenswert ist die canExecute-Methode der Handler. Kann der Handler nicht ausgeführt werden, die canExecute-Methode liefert also *false*, ist auch das Icon in der Toolbar nicht anklickbar. In Abbildung 4.5 ist dies beim Speichern der Fall.

4.1.3 Nachbereitung

Im Rahmen der zur Verfügung stehenden Zeit wurde ein Prototyp einer e4-Applikation umgesetzt, der die aufgelisteten Funktionen bereitstellt. Es ist möglich:

- Kontakte und Firmen anzulegen und dauerhaft in einer H2-Datenbank zu speichern
- zu bereits vorhandenen Firmen neue Kontaktpersonen anzulegen mit der Möglichkeit Adressangaben von bereits vorhandenen Kontakten (dieser Firma) zu übernehmen
- alle in der Datenbank vorhandenen Kontakte/Firmen im Suchergebnisfeld aufzulisten
- einzelne Kontakte oder Firmen per Doppelklick zu öffnen
- Kontakt und Firmendaten zu bearbeiten

Für die Möglichkeit des Löschens wurde der Rahmen gelegt, jedoch auf Implementierung im Datenservice und auf der Seite des Servers verzichtet. Dessen ungeachtet, konnte ein sehr guter Eindruck von der Entwicklung mit den e4-Technologien gewonnen werden. Der geringe Funktionsumfang lässt sich mit der mangelnden Erfahrung sowohl im Umgang mit den e4-Technologien als auch im Java Enterprise Umfeld begründen. Auch die fehlende Kenntnis der Oberflächenbibliothek SWT trug zum geringen Umfang bei. Es sei folgend auf die Schwerpunkte der Entwicklung eingegangen.

Implementiert wurde eine Client-Server-Architektur im Java Enterprise-Umfeld. Lediglich die Clientseite ist e4-spezifisch, das Backend mit EJBs und JBoss, welches ebenfalls mitentwickelt werden musste und nur einen geringen Beitrag zur Evaluierung beiträgt, nahm dennoch etwa ein Viertel der zweimonatigen Entwicklungszeit in Anspruch.

Ein weiteres Viertel der verfügbaren Zeit nahm die Einarbeitung in die SWT-Bibliothek und die Implementierung der GUI in Anspruch. Besonders aufwändig war die Einarbeitung in die GUI-Layouts.

Die andere Hälfte verblieb der Erkundung des sinnvollen Einsatzes der e4-Technologien. Durch die spärliche und unzusammenhängende Dokumentation

musste an dieser Stelle viel ausprobiert werden. Der effektive Einsatz von Annotationen und vorhandenen Services im Zusammenhang mit der Dependency Injection, wie im Listing 4.3 des SaveHandlers, glich häufig einem Puzzle, welches sich aus zahlreichen Erwähnungen und Andeutungen in Dokumentationen, Foren und Blogs zusammensetzte. Erwähnenswert sind hierbei das Eclipse e4-Wiki ,das Eclipse Forum, und die Blogs von Tom Schindel, Lars Vogel und Kai Tödter.[3]
Die e4-Plattform bietet mit der Möglichkeit der Dependency Injection eine komfortable Möglichkeit sich Daten, (aktive) grafische Komponenten oder Selections fast beliebig zu injizieren. Besonders Part- und SelectionService stellen gute Optionen dar, mit der Benutzeroberfläche und dem Applikationsmodell zu interagieren. Im Folgenden wird vor einer kurzen Zusammenfassung auf eine Problemstellung während der Entwicklung eingegangen.

Als ein Problem stellt sich die Injektion der benötigen Daten in die Part-POJOs heraus. Zwar ist es in diesen möglich, mittels injiziertem Daten Service an alle Daten zu gelangen. Dem POJO mitzuteilen welche Daten es anfordern soll, stellte sich als nicht trivial heraus. Grund dafür ist die lose Kopplung des POJOs mit dem Part im Modell. Dem Modell wird lediglich die Class-URI übergeben. Das POJO selbst hat keine Information darüber, welchem Part im Applikationsmodell es zugeordnet ist, in welchem Kontext es aufgerufen wurde und welche der möglichen Daten es laden soll. Die Eingabemaske aus Abbildung 4.6 soll als Beispiel zur Veranschaulichung dienen.

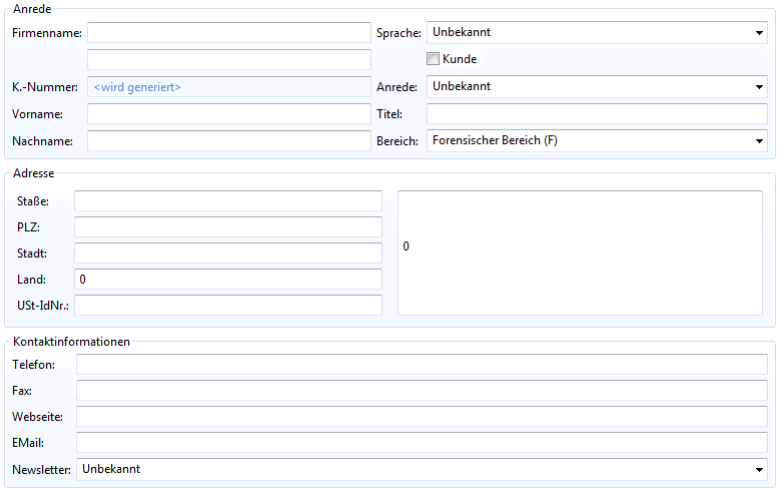

Abbildung 4.6: Eingabemaske des CustomerParts

[3]Vgl. [E4Wiki] [E4Foru] [SchBlog] [VogBlog] [ToeBlog]

Diese Maske für die Eingabe der Kontaktdaten befindet sich auf dem entsprechenden Reiter des `CustomerParts`. Die vollständige Implementierung des POJOs befindet sind im Anhang A.11. Im Kontaktmanager sind vier Fälle der Daten-Injektion zu differenzieren:

1. Ein neuer Kontakt soll angelegt werden. Leere Entitys müssen injiziert und mit der Maske verbunden werden.

2. Ein neuer Kontakt soll einer bestehenden Firma hinzugefügt werden. Die Company-Entity muss mit einer leeren User-Entity an die Maske gebunden werden.

3. Ein neuer Kontakt soll einer bestehenden Firma hinzugefügt werden, die Adresse soll übernommen werden. Die Company-Entity muss mit einer teils gefüllten User-Entity an die Maske gebunden werden.

4. Ein Kontakt soll bearbeitet werden. Die bestehenden Entitys müssen injiziert und an die Maske gebunden werden.

Die folgenden drei Lösungsansätze wurden in Erwägung gezogen:

1. Die spezifischen Handler erzeugen den Inhalt durch verschiedene Implementierungen des CustomerParts. Das heißt einem `NewCustomerPart`, `EditCustomerPart`, `NewWithCompanyPart` und `NewWithAddressPart`.
   Vorteile des Ansatzes sind die kaum benötigte Logik innerhalb des POJOs und der nicht vorhandene Aufwand dem POJO mitzuteilen, welche Daten geladen werden müssen. Als nachteilig stellt sich die hohe Redundanz heraus.

2. Die spezifischen Handler injizierten die nötigen Informationen in den einen CustomerPart. Da keine direkte Beziehung zwischen diesen besteht, wird dies per Events realisiert.
   Der Vorteil der Variante zwei ist die Reduzierung der POJOs auf eine Klasse. Für die Injektion der Information in den CustomerPart ist eine Erzeugung dieses POJOs in zwei Schritten nötig, zudem steigt der Anteil an Logik innerhalb des Part-POJOs.

3. Die spezifischen Handler teilen dem Datenservice mit, welche gleich angeforderten Daten er in den einen CustomerPart zu injizieren hat.
   Der dritte Lösungsvorschlag verknüpft zwar die Vorteile der beiden vorhergehenden, gleichzeitig verliert der Datenservice aber seine Unabhängigkeit. Neben der Datenbeschaffung wird der Service auch für die korrekte Verteilung der Daten zuständig.

In der neuen Version des Kontaktmanagers wurde sich für die zweite Variante der Lösungsvorschläge entschieden. Gründe, warum sich gegen den dritten Vorschlage entschieden wurde, sind entstehende Abhängigkeiten zwischen Handlern, Part-POJOs und Services, die vermieden werden sollen. Der erste Ansatz kommt durch die unnötige Redundanz nicht in Frage.
Die Umsetzung erfolgt mit der in Abschnitt 3.3.2 angedeuteten e4-Annotation `@UIEventTopic`. Dieser Annotation und ein `IEventBroker` können Events innerhalb einer e4-Applikation senden und empfangen. Codesnippets 4.5 und 4.6 verdeutlichen die Verwendung innerhalb des neuen Kontaktmanagers.

```
1  public class OpenCustomerHandler {
2    @Execute
3    public void execute(EModelService service, MApplication application,
4        EPartService partService, ESelectionService selectionService,
5        IEventBroker eventBroker) {
6    ...
7    MPart part=partService.createPart("com.qualitype.contact2.customerPart");
8    ...
9    partService.showPart(part, PartState.CREATE);
10
11   if(eventBroker.send(EventTopic.LOAD_CUSTOMER, id)){
12     partService.activate(part);
13     }
14   }
15 }
```

Listing 4.5: Code-Ausschnitt des Handlers zum Öffnen eines Kontakts

Der Handler erhält die ID des Kontaktes mit Hilfe des SelectionService. Weiterhin ist die Erzeugung des Parts in zwei Schritten zu erkennen. In Zeile 7 wird der PartDescriptor aus dem Applikationsmodell initialisiert und in Zeile 9 erzeugt.

```
1  @Inject @Optional
2  private void loadCustomer(@UIEventTopic(EventTopic.LOAD_CUSTOMER) Integer i)
3  {
4    if(i != null){
5      userEntity = dataService.getEntity(UserEntity.class, i);
6      companyEntity = userEntity.getCompany();
7      buildUI();
8    }
9  }
```

Listing 4.6: Methode des CustomerParts zum Laden der Daten

Würde der Part nun aktiviert, so ist die Eingabemaske leer. Daher wird mit Hilfe des EventBrokers das Event *EventTopic.LOAD_CUSTOMER*[4] ausgelöst, an welchem die ID des zu ladenden Kontaktes hängt. Zudem bewirkt die send Methode das Blockieren weiterer Schritte bis da sEvent empfangen wur-

[4]Dieses Event-Topic wurde im `ui` Plug-in des Kontaktmanagers implementiert.

de. An dieser Stelle bedeutet das bis die Daten in die loadCustomer-Methode injiziert und die Methode ausgeführt wurde.

4.1.4 Zusammenfassung

- Der neue Kontaktmanager besitzt ein simples Applikationsmodell, in dem der Aufbau der Anwendung gut modelliert werden kann. Dies betrifft sowohl UI-Komponenten, als auch Handler, Command, Bindings und PartDescriptors.
- Die POJOs der Parts werden bei aufwändigen UIs schnell unübersichtlich, eine Auslagerung von Teilen steigert die Übersichtlichkeit nur bedingt.
- Die POJOs der Handler sind dank der DI im Allgemeinen sehr kompakt. Die canExecute Methode ist, insofern vorhanden, häufig komplex.
- Es kommt häufig zur Vermischung von Logik und UI in den Part POJOs durch die DI.
- Die DI bietet viele Möglichkeiten. Die effektive Verwendung setzt jedoch gute Kenntnis vorhandener Annotations und Services voraus.
- Das Zusammenspiel von annotierten Methoden und der UI wirkt sehr durchdacht. Zum Beispiel die canExecute-Methode der Handler.
- Die Verwendung von PartDescriptors zur dynamischen Erzeugung von Parts ist empfehlenswert, da diese eine Übersicht im Applikationsmodell bieten, welche Parts erzeugt werden können. Das Anlegen der Descriptors im e4-Workbench Modelleditor ist im Eclipse SDK 4.1 jedoch insofern fehlerhaft, das Icon- und Class-URI manuell eingeben werden müssen.
- Die Warnung `Discouraged access` bei Nutzung von e4-Services und Annotationen ist laut Aussagen [Vogel10b] *... ok. The e4-project has not yet released their API therefore you receive warnings. The plan is to freeze the API in 4.1* Offensichtlich wurde dieser Plan vom November 2010 noch nicht in die Tat umgesetzt.
- Die Kommunikation von e4 und dem JBoss 6.1 war mittels implementiertem Datenservice einwandfrei.

4.2 Aufwand für Entwickler

Entscheidend für den generellen Aufwand einer Umstellung ist die Kenntnis des Eclipse RCP-Frameworks. So verringern e4 unabhängige Kenntnisse, wie die der SWT-Bibliothek oder der EJB-Entwicklung, den Einarbeitungsaufwand stark. Ausschlaggebend ist auch welche Art von Anwendung umgestellt werden soll. Eine serviceorientierte RCP-Applikation bedeutet ein Vielfaches weniger Aufwand, als eine monolithische Swing-Anwendung, da bei Portierung große Teile der bestehenden Businesslogik in OSGi-Services überführt werden. In beiden Fällen ist es notwendig das zentrale Applikationsmodell zu erzeugen. Dies ist, auch wenn das Modell nur eine grobe Abstraktion darstellt, je nach Komplexität der GUI sehr aufwändig. Das Ziel muss sein das Applikationsmodell als eine Art zentrales Nachschlagewerk nutzen zu können.
Der Einarbeitungsaufwand in e4-spezifische Annotationen und Services ist überschaubar, aber nicht zu unterschätzen.

Das Umstellen einer Applikation ist immer mit der Erstellung des Applikationsmodells verbunden. Zudem müssen POJOs für Parts und Handler implementiert werden. Selbst bei vorhandenen Handlern und RCP-GUI-Klassen stellt dies einen erheblichen Aufwand dar, da schlussendlich jede Klasse, wenn auch nur geringfügig, angepasst werden muss.

Ein weiterer Schwerpunkt ist der richtige Grad der Modularisierung. Wie umfangreich sind einzelne Services? Welche Teile des Applikationsmodells sollen in Contributions ausgelagert werden? Wie viel Logik soll in einem Part POJO stecken? Wie wird modularisiert, um möglichst viele Teile(Plug-ins) wieder zu verwenden? Umfangreiche Kenntnisse in der Eclipse RCP-Entwicklung sind, wie anfangs erwähnt, in diesem Zusammenhang von großem Vorteil.

4.3 Aufwand für Endanwender

Für den Endanwender entstehen bei der Umstellung einer Applikation keinerlei Nachteile. Durch das weniger starre System ohne Perspektiven und die Editor-View-Verschmelzung hat das grafische Nutzerinterface weniger IDE-Charakter. Die Bedienung ist identisch geblieben, eine Umgewöhnung ist damit nicht notwendig. Aussagen zur Performance-Verbesserung oder Verschlechterung können nicht getroffen werden. Die Vermutung liegt jedoch nahe, dass sich beim allzu sorglosen Umgang mit Injektionen, vor allem in rechenintensive Methoden, die Performance der Applikation verschlechtert.

4.4 Zukunft des Eclipse RCP-Frameworks 3.x

Laut der Aussage von [Sternb11] wird mit dem Release der Eclipse Version 4.2 im Juni 2012 dieses die primäre Version. Die Wikipediaseite der Eclipse IDE enthielt diese Information bis zum 29.10.2011 ebenfalls. Jedoch wurde diese mit dem Kommentar "*[...], warten wir doch ab, ob der termin gehalten wird*" wieder entfernt [WikiEcl]. Da hierzu keine zusätzlichen Informationen gefunden werden können, kann an dieser Stelle keine weitere Aussage zur Zukunft des Eclipse RCP-Frameworks 3.x getroffen werden.

4.5 Praxisorientierte Evaluierung

Analog zur Evaluierung auf Grundlage der theoretischen Kenntnisse soll das Eclipse RCP-Framework 4.1 erneut mit den neu erworbenen Kenntnissen auf die Produktionstauglichkeit evaluiert werden.

Die modulare, erweiterbare Plattform (Plug-in-Gliederung), welche das Framework bietet, wurde erweitert, wenngleich die grundlegenden Gliederung in Plug-ins unverändert blieb. Der stark ausgeprägte serviceorientierte Charakter des Frameworks zwingt förmlich zu einer Aufteilung der Applikation. Die Entwicklung innerhalb eines Teams wird durch eine mögliche Aufteilung des Applikationsmodells in Contributions weiter verbessert.

Die lose Kopplung der Module ist durch die intensive Nutzung von Services, der DI und POJOs von selbst gegeben. Bei der Implementierung ist dennoch stets auf ungewollte Abhängigkeiten zu achten.

Die Widgets, die im Prototyp verwendet wurden, entstammen alle der SWT Bibliothek. Auch wurde im Rahmen des Prototyps das Rendermodell unverändert gelassen. Bis auf die aufwändige Einarbeitung in die SWT Layouts, gab es seitens des Frameworks nur geringe Probleme, die jedoch mit dem CSS Styling in Verbindung stehen.

Die verwendeten Dialoge fügten sich einwandfrei in die e4-Umgebung ein. Besonders das Injizieren der aktiven Shell macht das Ausgeben von Information auf der GUI denkbar einfach.

Die Applikationsplattform in Form des Applikationsmodells stellt neben der DI die größte Verbesserung dar. Im Prototypen bot diese eine hervorragende Übersicht über die Applikation. Auch bei sehr umfangreichen Applikationen sollte diese dank Contributions gewährleistet bleiben. Von dieser Möglichkeit wurde durch den geringen Umfang des Modells im Prototypen kein Gebrauch gemacht.

Der als Unterstützung dienende Editor des Applikationsmodells wurde bei der Entwicklung ausgiebig genutzt. Wünschenswert wäre zudem eine Unterstützung zur Dependency Injection gewesen, etwa in Form einer Übersicht, die veranschaulicht was aktuell injiziert werden kann. In dieser Übersicht sollten dann ebenfalls die verfügbaren Services aufgelistet sein. Bei umfangreichen Projekten mit vielen Entwicklern ist diese fehlende Übersicht möglicherweise von Nachteil.

Die stabile und fehlerfreie Funktion von angebotenen Features stellt sich auch bei der Umstellung des Kontaktmanagers als Problem heraus. So ist der Workbench Modelleditor der Part Descriptors fehlerhaft, in der Weise, dass keine Class- und Icon-URIs ausgewählt werden können. Da auch ein Einfügen aus der Zwischenablage nicht möglich ist, muss der lange Pfadname per Hand eingeben werden. Dies ist bei einer Umstellung komplexer Anwendungen inakzeptabel. Weiterhin ist es nicht möglich eine Toolbar-Contribution dem Applikationsmodell hinzufügen. Als störend wirkt zudem die allgegenwärtige Warnung des *Discouraged access* bei der Verwendung von e4-Bibliotheken. Der aus dem Eclipse Framework 4.1 nicht zu startende JBoss 6 ist ein weiteres Ärgernis. Zwar funktioniert das Deployment tadellos, jedoch erkennt die IDE den korrekt gestarteten JBoss nicht als solchen an. Weitere bekannte Fehler sind unter [E4Bugs] aufgelistet.

Wesentlich weniger Boilerplate-Code ist durch die DI gewährleistet. Vor allem die Handler POJOs sind angenehm schlank. Die geschickte Kombination von Applikationsmodell, vorhandenen Services und der DI machen dies möglich. Voraussetzung ist die Kenntnis der Services und Annotationen, sowie deren Zusammenspiel mit dem Applikationsmodell.

Der Charakter der GUI ist wie an den Screenshots zu erkennen von einer IDE weit entfernt, selbst wenn das Styling noch nicht völlig ausgereift ist. So ist die Extension-Bar der Suchdetails in Abbildung 4.5 keinem Styling unterworfen

und ist dennoch seltsam grau. Als angenehmer Nebeneffekt des Stylings mithilfe einer CSS Datei, ist es möglich dieses jeder anderen e4-Applikation leicht zur Verfügung zu stellen. Im Falle des Standard Namen *default.css* reicht ein Copy-Paste-Vorgang, um einer anderen Applikation das gleiche Aussehen zu verleihen. Neben dem eigentlichen Styling wirken sich zu dem die optionalen Perspektiven und das Part Konzept positiv auf das Aussehen der Applikationen aus.

Die erste Evaluierung zeigte, dass das Eclipse 4.1 RCP-Framework produktionstauglich ist. Mit der Umstellung des Kontaktmanagers konnten viele praktische Erfahrungen mit dem Framework gesammelt werden. Wie anhand der Kriterien zu erkennen ist, überzeugen die Neuerungen des Frameworks auch in der konkreten Anwendung – jedoch nicht durchweg. Die stabile und fehlerfreie Funktion der IDE-Features, die von den Entwicklern gefordert wurde, kann an einigen Stellen nicht gewährleistet werden. Besonders schwer wiegt hier der Fehler im Editor der Part Descriptors und das fehlerhafte Zusammenspiel der IDE mit installierten Plug-ins, wie den JBoss Tools. Auch bei der Unterstützung für die Entwicklung mit dem neuen Programmiermodell werden die Entwickler enttäuscht.

Da im Rahmen dieser Arbeit keine komplexe, bestehende Anwendung umgestellt werden konnte, wurde stattdessen auf eine angemessene Applikation zurückgegriffen. Bereits bei der Erstellung einer Applikation dieser Größenordnung waren die fehlerhaften Funktionen der IDE störend. Entsprechend schwerer wiegen diese bei der Umstellung komplexer Anwendungen. Trotz des großen Potenzials der Neuerungen, kann die Produktionstauglichkeit in der praxisorientierten Evaluierung des Eclipse 4.1 RCP-Frameworks nicht gewährleistet werden.

5

Resultate und Diskussion

5.1 Ergebnisse und Auswertung

Evaluierung der Produktionstauglichkeit

Das Ziel des ersten Teils der Arbeit war die Evaluierung des Eclipse RCP-Framworks 4.1 auf Produktionstauglichkeit. In den beiden durchgeführten Evaluierungen überwiegen die Verbesserungen durch das Framework. Jedoch existieren Kritikpunkte, die vor allem in der praktischen Nutzung, das heißt der Entwicklung mit dem Framework schwer wiegen.
Die von den Entwicklern geforderte, fehlerfreie und stabile Unterstützung der von der IDE angebotenen Features ist nicht gewährleistet. So ist der Editor des Applikationsmodells bei der Erweiterung des Rendermodell nicht mehr nutzbar. Da dieses ein zentrales Bestandteil der Applikation darstellt und die Entwicklung zwingend von der Verfügbarkeit des Editors abhängt, ist die IDE damit völlig nutzlos. Die fehlerhafte Funktionalität bei der URI-Auswahl der Part-Desciptors, stellt zwar einen weniger kritischen Ausfall dar, dennoch ist dieser bei komplexen Applikationen inakzeptabel.

Das Gesamtergebnis der Evaluierung ist konsequenterweise die nicht gewährleistete Produktionstauglichkeit des Eclipse RCP-Frameworks 4.1. Trotz der negativen Evaluierung sei die Entwicklung von e4-Applikationen empfohlen. Das Eclipse RCP-Framework 4.1 bietet mit dem Applikations- und dem Pro-

grammiermodell eine hervorragende Grundlage für die Entwicklung von plug-inbasierten und plattformunabhänigen Java Enterprise-Applikationen der nächsten Generation. Die Tabelle 5.1 fasst die Ergebnisse der Evaluierungen zusammen. Die grünen Symbole stehen dabei für erfüllt/Verbesserung, die gelben für teilweise erfüllt/fehlerhaft/keine Verbesserung und die roten Symbole für nicht erfüllt/schwerwiegende Fehler.

Tabelle 5.1: Zusammenfassung der Evaluierungsergebnisse

Evaluierungskriterium	**theoretisch**	**praktisch**	**gesamt**
modulare und erweiterbare Plattform	✚	✚	✚
lose Kopplung der Module	✚	✚	✚
native, performante Widgets	▬	▬	▬
standardisiertes Dialog- und Wizard-Framework	▬	✚	✚
Applikationsplattform, die den Rahmen der Anwendung sowie deren Laufzeitumgebung festlegt	✚	✚	✚
IDE Unterstützung neuer Framework Features	▬	▬	▬
stabile, fehlerfreie Funktion von angebotenen Features der IDE	▬	✖	✖
wenig Boilerplate-Code	✚	✚	✚
IDE Charakter der GUI der Applikation unerwünscht	✚	▬	✚
Gesamtergebnis der Evaluierung	✚	✖	✖

Aufwandsabschätzung

Die grundlegenden Aussagen für die Abschätzung des Aufwands finden sich im Abschnitt 2 und 3 des vierten Kapitels. Der Aufwand der Umstellung einer komplexen Applikation hängt stark von der vorhandenen Anwendung ab. Ist die umzustellende Applikation eine RCP-Anwendung, kann der Aufwand geringer ausfallen, abhängig davon wie serviceorientiert diese ist. Ist die Applikation nicht aus dem RCP-Umfeld wird die Umstellung einer Neuentwicklung gleichen.
Neben der Einarbeitung in die Entwicklung von e4-Applikationen ist die Erstellung des zentralen Applikationsmodells obligatorisch. Soll die vorhandene Anwendung nur innerhalb einer e4-Umgebung gestartet werden, ist dies nicht

notwendig. Da es sich nicht um eine Umstellung handelt, wird dieser Fall hier nicht berücksichtigt. Der folgenden Punkte werden bei jeder Umstellung vollzogen.

- Erstellen eines zentralen Applikationsmodells
- Anpassen oder Erstellen von Handler- und Part-POJOs
- Anpassen oder Erstellen von Services
- Anpassen oder Erstellen von Contributions
- Verknüpfen des Modells mit POJOs , Services und Contributions

Entsprechend der umzustellenden Applikation wird mehr oder weniger Aufwand für einzelne Punkte nötig sein. Wie an den Punkten zu erkennen ist, wird der Umstellungsaufwand einer serviceorientierten Applikation im RCP-Umfeld der geringste sein, da in diesem Fall mehr angepasst als erstellt werden muss. Über die Qualität der resultierenden Applikation bei der Umstellung im Vergleich zur Neuentwicklung kann an dieser Stelle keine Aussage getroffen werden. Der Aufwand wird in jedem Falle höher ausfallen.

5.2 Diskussion

Grundsätzliches Problem war die mangelnde Erfahrung im Bereich der Eclipse-RCP-Entwicklung. Daher musste sich nicht nur in das Framework 4.1 eingearbeitet werden, sondern generell in die RCP-Welt. Vorteilhaft ist jedoch die Unvoreingenommenheit von der Entwicklung mit dem Eclipse 3.x Framework. Der Einarbeitungsaufwand war dadurch jedoch höher.

Bei der Umstellung einer Applikation kam die Tatsache erschwerend hinzu, dass neben der e4-Client Entwicklung auch das Backend mit Enterprise JavaBeans und JBoss-Anbindungen entwickelt werden musste. Der Umgang mit der SWT-Bibliothek und den resultierenden Databinding-Konzepten musste ebenfalls angeeignet werden. Die dafür benötigten Aufwände reduzierten entsprechend den Funktionsumfang der umgestellten Applikation.

Die Arbeit erfüllt die beiden Ziele des Themas vollständig. Der Abschnitt 5.1 legt dabei die Ergebnisse und deren Auswertung dar. Im gesamten Umfang kann die Arbeit auch als Einführung in die Entwicklung von e4-Applikationen interpretiert werden. Das Kapitel 3 kann dabei als Nachschlagewerk und das Kapitel 4 als praktisches Beispiel angesehen werden. Insbesondere die detaillierten Grafiken können als Gedächtnisstütze dienen. Neben der entwickelten Applikation befinden sich diese Grafiken daher in verschiedenen Formaten zum Ausdrucken auf der beigefügten CD. Die zahlreichen Codebeispiele im Anhang sind in diesem Zusammenhang ebenfalls nützlich.

5.3 Ausblick

Das Eclipse RCP-Framework 4.1 wurde innerhalb dieser Arbeit ausgiebig behandelt. Dabei lag der Schwerpunkt mehr auf dem Aufzeigen der zur Verfügung stehenden Möglichkeiten und weniger darauf wie diese in komplexen Anwendungen ihre optimale Wirkung entfalten. Ziel einer weiteren Arbeit könnten Architekturmuster und Best Practices im Umgang mit Services und der Dependency Injection sein. Dies betrifft sowohl das Frontend mit der Injizierung von Daten in die Benutzeroberfläche, die serviceorientierte Middleware, als auch das Backend mit EJBs und Application-Server. Ein Ziel dieser möglichen Arbeit könnte die Entwicklung von Applikationen im e4-Umfeld sein, deren Service-Plug-ins eine besonders hohen Wiederverwendungswert haben, wie ein applikationsunabhängiger Login-Service.

Anhang

A

Anlagen

A.1 Installationshinweise E4 Tools

Um an die benötigten Tools zu gelangen, ist unter Help->Install new Software, die Updateseite *e4 0.11 |Build Updates* auszuwählen, und unter E4 Tools die Eclipse e4 Tools (Incubation) zu installieren. Die anderen Tools sind optional.

Anschließend Eclipse neu starten.

Die Tools beinhalten neben Wizards zum Erzeugen einer e4-Applikation, den e4-Workbench Modelleditor. Ohne diesen ist keine effektive Entwicklung von e4-Anwendungen möglich.

A.2 Dependency Injection Annotations

Die detaillierten Beschreibungen sind unter 3.3.2 zu finden.

Annotation	Java-Package	Erläuterung
@Inject	javax.inject	Markiert Felder, Methoden oder Konstruktoren, welche für die DI verfügbar sind.
@Named	javax.inject	Stringbasierter Bezeichner für Felder und Parameter. In Verbindung mit @Inject.
@PostConstruct	javax.annotaion	Lifecycle-Management. Aufruf der annotierten Methode nach vollständiger Injektion.
@PreDestroy	javax.annotaion	Liefecyle-Management. Aufruf vor Löschen des Objektes(aus dem Kontext).
@Optional	*.e4.core.di.annotations	Optionale Injektion(mit Fehlerbehandlung).
@Active	*.e4.core.context	Injektion einer aktiven Komponente. (z.B. Aktiven Part)
@Preference	*.e4.core.di.extensions	Injektion einer Preference aus dem Eclipse Framework.
@CanExecute	*.e4.core.di.annotations	Optionale Voraussetzung für @Execute innerhalb eines Handler. (boolean-Returnwert!)
@Execute	*.core.di.annotaions	Markierung einer auszuführenden Methode eines Handlers.
@Focus	org.eclipse.e4.ui.di	Markierte Methode wird beim Erhalt des Fokus aufgerufen.
@GroupUpdates	*.e4.core.di.annotations	Stapelweise/blockweise Injektion mehrerer Felder.
@UIEventTopic	org.eclipse.e4.ui.di	EventTopics, Verwendung zum Zusammenhang mit einem IEventBroker, der entsprechende Events wirft.
@Persist	org.eclipse.e4.ui.di.	Markiert Methoden in Parts aus denen heraus gespeichert wird. Aufruf über die savePart()-Methode des EPartService.

*org.eclipse

```
@Inject
@Named(E4Workbench.INSTANCE_LOCATION)
private Location instaceLocation;
// oder auch
@Inject @Named("string") private String s;
```

Listing A.1: Implementierung @Named Annotation

```
@Inject
@Optional @Named("string")
private String optionalString;

@Inject
private void setSelection(@Optional Selection selection) {
  if(selection != null) ...
}

@Inject
@Optional
private void setSelection(Selection selection) {
  ...
}
```

Listing A.2: Implementierung @Optional Annotation

```
@Inject
private void doSomething(@Optional @Active MPart activePart) {
   ...
}
```

Listing A.3: Implementierung @Active Annotation

```
@Inject @Preference(node="de.preinhold.example.di", value="dateFormat")
private String dateFormat;

@Inject
private void setDateFormat(@Preference(node="my.plugin.id",
 value="dateFormat") String dateFormat) {
    ...
}

@Inject @Preference(node="de.preinhold.example.di")
IEclipsePreferences preferences;
private void doSomething() {
    preferences.put(dataFormat, ...);
}
```

Listing A.4: Implementierung @Preference Annotation

```
public Handler{
  @Execute
  public void execute(IWorkbench workbench) {
  ... // Das workbench-Object wird injiziert
}
  @CanExecute
  public boolean canExecute() {
  if(...)
}
```

Listing A.5: Implementierung @Execute und @CanExecute Annotation

```
public Handler{
  @Execute
  public void execute(IWorkbench workbench) {
  ... // Das workbench-Object wird injiziert
}
  @CanExecute
  public boolean canExecute() {
  if(...)
}
```

Listing A.6: Implementierung @UIEventTopic Annotation

```
public Handler{
  @Execute
  public void execute(IWorkbench workbench) {
  ... // Das workbench-Object wird injiziert
}
  @CanExecute
  public boolean canExecute() {
  if(...)
}
```

Listing A.7: Implementierung @Persist Annotation

Beispielimplementierung der in 3.3.2 vorgestellten Annotationen.

```
1  public ExamplePart {
2
3  //1. Inject
4  @Inject private Logger logger;
5  @Inject private Composite parent;
6
7    //2. PostContruct
8    @PostConstruct
9    private void buildUI() {
10     logger.info("Building UI ...");
11     Label lab = new Label(parent, SWT.NONE);
12     lab.setText("ExamplePart");
13     ...
14   }
15
16   @PreDestroy
17   private void cleanUp() {
18     logger.info("Cleaning up ...");
19     ...
20   }
21
22   @Focus
23   private void handleFocus() {
24     logger.info("Got Focus ...");
25     ...
26   }
27
28   @Persist
29   private void save() {
30   logger.info("Save ExamplePart ...");
31    ...
32   }
33
34 }
```

Listing A.8: Vereinfachtes Beispiel Dependency Injection (Part)

A.3 Application.e4xmi Addons

Die aufgelisteten Addons stehen zur Verfügung. Die entsprechenden Code-Schnipsel können direkt vor dem schließenden Application-Tag in die Application.e4xmi kopiert werden. (Was ist mit den IDs?)

Command Service Addon

- für jede e4 Applikation notwendig

```
<addons xmi:id="_qbWepHukEeCRy6IAJt6DXA"
elementId="org.eclipse.e4.core.commands.service"
contributionURI="platform:/plugin/org.eclipse.e4.core.commands/
org.eclipse.e4.core.commands.CommandServiceAddon"/>
```

Command Processing Addon

- für jede e4 Applikation notwendig

```
<addons xmi:id="_qbWep3ukEeCRy6IAJt6DXA"
elementId="org.eclipse.e4.ui.workbench.commands.model"
contributionURI="platform:/plugin/org.eclipse.e4.ui.workbench/
org.eclipse.e4.ui.internal.workbench.addons.CommandProcessingAddon"/>
```

Context Service Addon

- für jede e4 Applikation notwendig

```
<addons xmi:id="_qbWepXukEeCRy6IAJt6DXA"
elementId="org.eclipse.e4.ui.contexts.service
 contributionURI="platform:/plugin/org.eclipse.e4.ui.services/
org.eclipse.e4.ui.services.ContextServiceAddon"/>
```

Binding Service Addon

- für jede e4 Applikation notwendig

```
<addons xmi:id="_qbWepnukEeCRy6IAJt6DXA"
elementId="org.eclipse.e4.ui.bindings.service"
 contributionURI="platform:/plugin/org.eclipse.e4.ui.bindings/
org.eclipse.e4.ui.bindings.BindingServiceAddon"/>
```

Context Processing Addon

- für jede e4 Applikation notwendig

```
<addons xmi:id="_qbWeqHukEeCRy6IAJt6DXA"
elementId="org.eclipse.e4.ui.workbench.contexts.model"
 contributionURI="platform:/plugin/org.eclipse.e4.ui.workbench/
org.eclipse.e4.ui.internal.workbench.addons.ContextProcessingAddon"/>
```

Binding Processing Addon

- für jede e4 Applikation notwendig
- erzeugt unter 4.1 M6 eine Nullpointer Exception (`https://bugs.eclipse.org/bugs/show_bug.cgi?id=340508`)

```
<addons xmi:id="_qbWeqXukEeCRy6IAJt6DXA"
elementId="org.eclipse.e4.ui.workbench.bindings.model"
 contributionURI="platform:/plugin/org.eclipse.e4.ui.workbench.swt/
org.eclipse.e4.ui.workbench.swt.util.BindingProcessingAddon"/>
```

CleanUp Addon

-optional

```
<addons xmi:id="_XwQYkE2EEd-DfN2vYY4Lew" elementId="Cleanup Addon"
contributionURI="platform:/plugin/org.eclipse.e4.ui.workbench.addons.swt/
org.eclipse.e4.ui.workbench.addons.cleanupaddon.CleanupAddon"/>
```

MinMax Addon

- ermöglicht das Minimieren und Maximieren von Parts

```
<addons xmi:id="_7GC6sGp-Ed-QyNZjH9g15Q" elementId="MinMax Addon"
 contributionURI="platform:/plugin/org.eclipse.e4.ui.workbench.addons.swt/
org.eclipse.e4.ui.workbench.addons.minmax.MinMaxAddon"/>
```

Drag and Drop Addon

- ermöglicht die Verwendung von Drag and Drop

```
<addons xmi:id="_bqcWME2EEd-DfN2vYY4Lew" elementId="DnD Addon"
 contributionURI="platform:/plugin/org.eclipse.e4.ui.workbench.addons.swt/
org.eclipse.e4.ui.workbench.addons.dndaddon.DnDAddon"/>
```

A.4 IEclipseContext Parameter- und Serviceübersicht

Workbench Services sind in aller Regel nach dem E*Service Muster benannt.

E4 Application Context

Application Parameters:

```
1  applicationCSS (E4Workbench#CSS_URI_ARG)
2  applicationCSSResources (E4Workbench#CSS_RESOURCE_URI_ARG)
3  applicationXMI (E4Workbench#XMI_URI_ARG)
4  clearPersistedState (E4Workbench#CLEAR_PERSISTED_STATE)
5  deltaRestore (E4Workbench#DELTA_RESTORE)
6  cssTheme (E4Application#THEME_ID)
7  initialWorkbenchModelURI (E4Workbench#INITIAL_WORKBENCH_MODEL_URI)
8  instanceLocation (E4Workbench#INSTANCE_LOCATION)
9  persistState (E4Workbench#PERSIST_STATE)
```

Services:

```
1  org.eclipse.core.databinding.observable.Realm
2  org.eclipse.core.runtime.dynamichelpers.IExtensionTracker
3  org.eclipse.core.runtime.IExtensionRegistry
4  org.eclipse.core.runtime.Platform
5  org.eclipse.e4.core.commands.ECommandService
6  org.eclipse.e4.core.commands.EHandlerService
7  org.eclipse.e4.core.services.adapter.Adapter
8  org.eclipse.e4.core.services.events.IEventBroker
9  org.eclipse.e4.core.services.log.Logger
10 org.eclipse.e4.core.services.translation.TranslationService
11 org.eclipse.e4.ui.css.swt.theme.IThemeEngine
12 org.eclipse.e4.ui.services.IStylingEngine
13 org.eclipse.e4.ui.workbench.IPresentationEngine
14 org.eclipse.e4.ui.workbench.IResourceUtilities
15 org.eclipse.e4.ui.workbench.modeling.EModelService
16 org.eclipse.equinox.app.IApplicationContext
17 org.eclipse.jface.preference.PreferenceManager
18 org.eclipse.ui.ISharedImages
19 org.eclipse.ui.progress.IProgressService
```

Runtime Data:

```
1  activePart (IServiceConstants#ACTIVE_PART)
2  org.eclipse.e4.core.locale (TranslationService# LOCALE)
3  org.eclipse.e4.ui.model.application.MApplication
4  selection (ESelectionService#SELECTION)
5  IServiceConstants.ACTIVE_SHELL
```

Top Level Window Context

Model Info: * Jeder, als MContext Element, erzeugte Kontext, fügt **alle** implementieren Interfaces zu seinem eigenen Kontext hinzu.

```
1 org.eclipse.e4.ui.model.application.commands.MBindings
2 org.eclipse.e4.ui.model.application.commands.MHandlerContainer
3 org.eclipse.e4.ui.model.application.MApplicationElement
4 org.eclipse.e4.ui.model.application.ui.basic.MTrimmedWindow
5 org.eclipse.e4.ui.model.application.ui.basic.MWindow
6 org.eclipse.e4.ui.model.application.ui.MContext
7 org.eclipse.e4.ui.model.application.ui.MElementContainer
8 org.eclipse.e4.ui.model.application.ui.MUIElement
9 org.eclipse.e4.ui.model.application.ui.MUILabel
```

Services:

```
1 org.eclipse.e4.ui.workbench.modeling.ESelectionService
2 org.eclipse.e4.ui.workbench.modeling.ISaveHandler
3 org.eclipse.e4.ui.workbench.modeling.EPartService
```

Runtime Data:

```
1 activePart (IServiceCOnstants#ACTIVE_PART)
2 selection (ESelectionService#SELECTION)
```

Part Context

Model Info:

```
1  org.eclipse.e4.ui.model.application.commands.MBindings
2  org.eclipse.e4.ui.model.application.commands.MHandlerContainer
3  org.eclipse.e4.ui.model.application.MApplicationElement
4  org.eclipse.e4.ui.model.application.MContribution
5  org.eclipse.e4.ui.model.application.ui.basic.MPart
6  org.eclipse.e4.ui.model.application.ui.basic.MPartSashContainerElement
7  org.eclipse.e4.ui.model.application.ui.basic.MStackElement
8  org.eclipse.e4.ui.model.application.ui.basic.MWindowElement
9  org.eclipse.e4.ui.model.application.ui.MContext
10 org.eclipse.e4.ui.model.application.ui.MDirtyable
11 org.eclipse.e4.ui.model.application.ui.MUIElement
12 org.eclipse.e4.ui.model.application.ui.MUILabel
```

Services:

```
1 org.eclipse.e4.ui.workbench.modeling.EPartService
```

A.5 CSS Style

Auswahl über vorhandene Selektoren

Selektor	Bemerkung
Tree	Ermöglicht Zugriff auf den TreeViewer
Label	Zugriff auf den Label(Text)
Composite	Zugriff auf die Gesamtstruktur, oberste Ebene der UI.
Text	Zugriff auf die Texteingabefelder
ToolBar	Zugriff auf die Toolbar(unter 4.x nur die Items, aber alle), keine Auswirkung von color auf die Schriftfarbe
CTabItem	Zugriff auf die PartStack Tabs, Alternative zu .MPartStack (auch ohne CTabRenderer)
CTabItem:selected	Zugriff auf den ausgewählten Tab
CTabItem.active	Alternative zu .MPartStack.active, (CTabRenderer notwendig)
CTabFolder	Hintergrund des MPartStacks (auch ohne CTabRenderer)
.MTrimmedWindow.topLevel	(nur 4.x) gesamtes Fenster (nur einmaliger Zugriff beim Start der Applikation ?) vergleichbar mit Composite?
.MTrimmedWindow	(nur 4.x) gesamtes Fenster, unter .topLevel angesiedelt
.MTrimBar	(nur 4.x) gesamte Trimbar, keine Auswirkungen mit color auf die Schriftfarbe
.MPart	(nur 4.x) Zugriff auf die Parts, den Inhalt der Tabs
.MPartStack	(nur 4.x) Zugriff auf die PartStack Tabs, inkl. Part wenn dieser nicht gesetzt. Schmaler Rand um Parts
.MPartStack.active	(nur 4.x) Zugriff auf den aktiven PartStack

Beispiel CSS *Contrast Color Thema*

```
.MTrimmedWindow.topLevel {
  margin-top: 12px;
  margin-bottom: 2px;
  margin-left: 5px;
  margin-right: 5px;
}
.MTrimmedWindow {
  background-color: cyan;
}
.MTrimBar {
  background-color: blue;
}
.MPart {
  background-color: yellow;
}

.MPartStack {
  tab-renderer: null;
  background-color: red;
  selected-tabs-background: gray 100%;
  simple: false;
}

.MPartStack.active {
  selected-tabs-background: green 100%;
}

Label {
  font-size: 12px;
  color: black;
}

#officeLabel {
  font-size: 8px;
  color: blue;
  font-weight: bold;
}

Composite Text {
  background-color: white;
  color: red;
}

#IdText {
  background-color: black;
  color: white;
}

Tree {
  background-color: gray;
  color: orange;
  font: Courier 6px;
}
```

Listing A.9: Contrast Color CSS

Screenshot CSS Theme

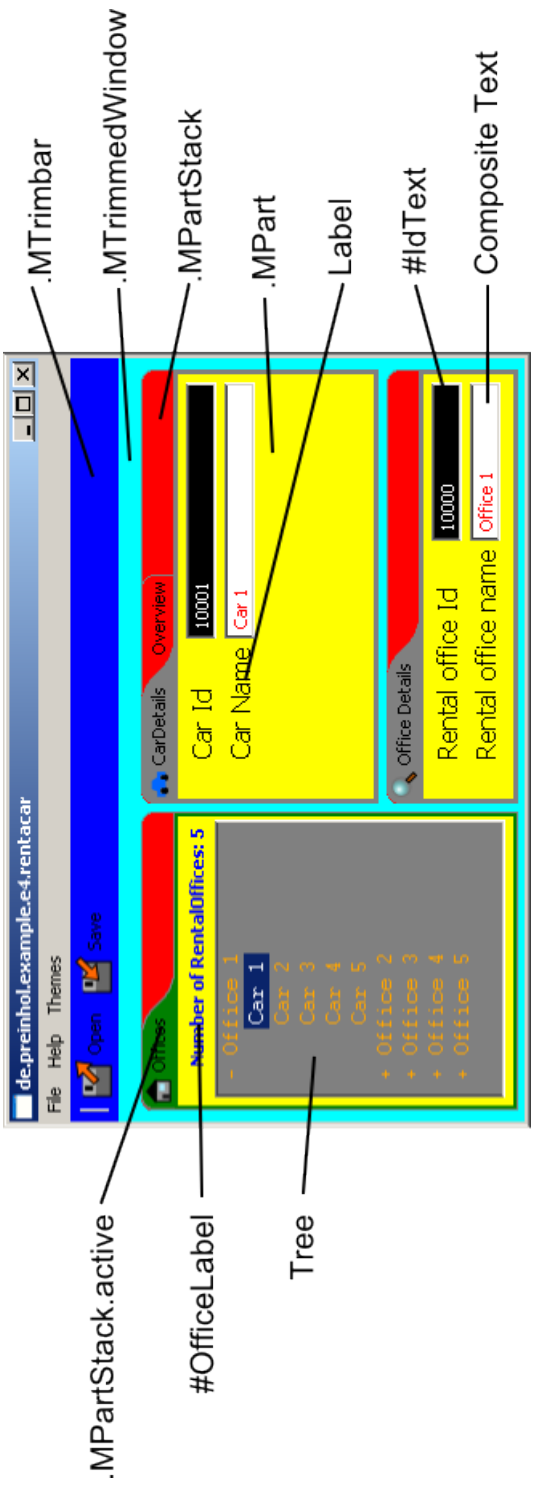

Abbildung A.1: Screenshot eines Contrast Color Themas

Gradienten Übersicht

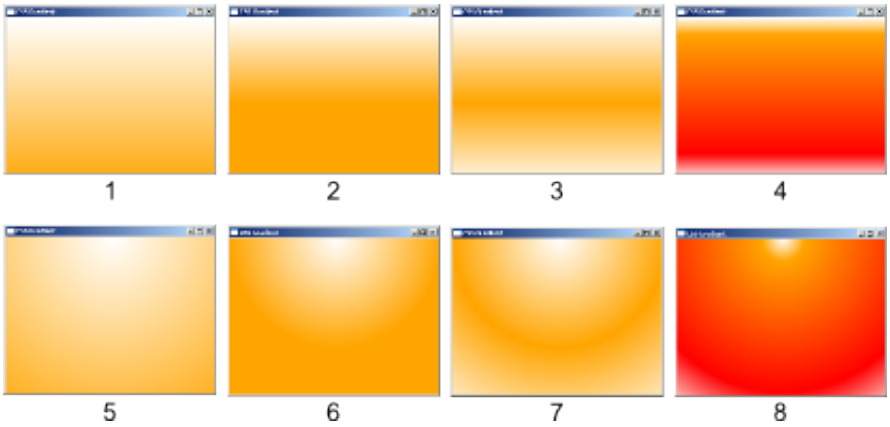

Abbildung A.2: Überblick über verschiedene Beispielgradienten.

Tabelle A.1: Beschreibgung der Gradienten aus Abbildung A.2. Der Codeausschnitt beschränkt sich auf Grund der Übersichtlichkeit auf den *Wert*, des Values `background-color`.

Nr.	**Art**	**CSS-Codeausschnitt**
1	linear	white orange 100%
2	linear	white orange 50%
3	linear	white orange white 50% 100%
4	linear	white orange red white 10% 70% 100%
5	radial	gradient radial white orange 100%
6	radial	gradient radial white orange 50%
7	radial	gradient radial white orange white 50% 100%
8	radial	gradient radial white orange red white 10% 50% 100%

A.6 Beispiel Implementierung eines Parts

```
1  public class MyPart {
2
3    @Inject private MDirtyable dirtyable;
4
5    private UserEntity userEntity;
6    private Text userName;
7    private DataService dataService;
8
9    @Inject
10   public MyPart(DataService dataService) {
11     this.dataService = dataService;
12     ...
13     userEntity = dataService.getUserEntity(fistName,lastName);
14     ...
15     DataBindingContext bindingContext = new DataBindingContext();
16     IObservableValue myModel =
17       PojoProperties.value(UserEntity.class, "name").observe(userEntity);
18     IObservableValue myWidget=
19       WidgetProperties.text(SWT.Modify).observe(userName);
20
21     myModel.addValueChangeListener(new IValueChangeListener() {
22       @Override
23       public void handleValueChange(ValueChangeEvent event) {
24         dirtyable.setDirty(true);
25       }
26     });
27     bindingContext.bindValue(myModel, myWidget);
28     ...
29   }
30
31   @Persist
32   private void save(@Active MPart activePart,
33     @Active Shell activeShell){
34
35     if(dataService.saveEntity(userEntity)) {
36       dirtyable.setDirty(false);
37       activePart.setLabel(userEntity.getLastName()+", "+
38         userEntity.getFirstName());
39       activePart.setIconURI("platform:/plugin/com.qualitype.contact2/"+
40         "icons/customer.png");
41     }
42     else {
43       MessageDialog.openError(activeShell, "Fehler", "Fehlerbeschreibung");
44     }
45   }
46 }
```

Listing A.10: Implementierung MyPart POJO

A.7 Implementierung des CustomerParts im Kontaktmanager

```
1  public class CustomerPart {
2    @Inject private MDirtyable dirtyable;
3    @Inject private IEventBroker eventBroker;
4    @Active private Shell activeShell;
5
6    private Composite parent;
7    private Display display;
8
9    //Entitys
10   private UserEntity userEntity;
11   private CompanyEntity companyEntity;
12
13   //Services
14   private DataService dataService;
15
16   @Inject
17   public CustomerPart(Composite parent,Display display,DataService dService)
18   {
19     this.parent = parent;
20     this.display = display;
21     this.dataService = dService;
22   }
23
24   @PostConstruct
25   private void buildUI(){
26     parent.setLayout(new FillLayout());
27     CTabFolder folder = new CTabFolder(parent, SWT.BOTTOM);
28     folder.setUnselectedCloseVisible(false);
29     folder.setSimple(true);
30     folder.setLayout(new FillLayout());
31
32     CustomerTabItem customerDataTab = new CustomerTabItem(folder,
33       SWT.NONE,userEntity,companyEntity,dirtyable);
34     HistoryTabItem historyTab = new HistoryTabItem(folder, SWT.NONE,display
35       ,userEntity,eventBroker, activeShell);
36     FileTabItem fileTab = new FileTabItem(folder, SWT.NONE, display);
37     MiscellaneousTabItem miscTab = new MiscellaneousTabItem(folder,
38       SWT.NONE, display);
39
40     folder.setSelection(customerDataTab.getCTabItem());
41   }
42
43   @Persist
44   private void save(@Active MPart activePart, @Active Shell activeShell,
45           IProgressMonitor progressMonitor)
46   {
47     //Speichern
48     dataService.saveEntity(companyEntity);
49     CompanyEntity persitComp = (CompanyEntity)
50       dataService.getCompanyEntity_READ_ONLY(companyEntity.getName());
51     userEntity.setCompany(persitComp); // FK setzen !
52     dataService.saveEntity(userEntity);
53
54     //Rest anpassen
```

```
55      dirtyable.setDirty(false);
56      activePart.setLabel(userEntity.getLastName()+",
57        "+userEntity.getFirstName());
58      activePart.setIconURI("platform:/plugin/com.qualitype.contact2/"+
59        "icons/customer.png");
60      activePart.setElementId(userEntity.getLastName()+",
61         "+userEntity.getFirstName());
62    }
63
64    @Inject
65    private void loadNew(@Optional
66        @UIEventTopic(EventTopic.NEW_CUSTOMER) Integer id)
67    {
68      userEntity = new UserEntity();
69      companyEntity = new CompanyEntity();
70    }
71
72    @Inject
73    private void loadCustomer(@Optional
74        @UIEventTopic(EventTopic.LOAD_CUSTOMER) Integer id)
75    {
76      if(id != null){
77        userEntity =
78          (UserEntity)dataService.getEntity(userEntity,id.intValue());
79        companyEntity = userEntity.getCompany();
80      }
81    }
82
83    @Inject
84    private void loadCompany(@Optional
85        @UIEventTopic(EventTopic.LOAD_COMPANY) Integer id)
86    {
87      if(id != null){
88        companyEntity =
89          (CompanyEntity)dataService.getEntity(companyEntity, id.intValue());
90      }
91    }
92
93    @Inject
94    private void loadAdress(@Optional
95        @UIEventTopic(EventTopic.LOAD_COMPANY_AND_CUSTOMER_ADDRESS)Integer id)
96    {
97      if(id != null){
98        UserEntity adressEntity =
99            (UserEntity)dataService.getEntity(userEntity,id.intValue());
100       companyEntity = userEntity.getCompany();
101
102       //userEntity anpassen
103       userEntity = new UserEntity();
104       userEntity.setStreet(adressEntity.getStreet());
105       userEntity.setCity(adressEntity.getCity());
106       userEntity.setZipCode(adressEntity.getZipCode());
107       userEntity.setCountry(adressEntity.getCountry());
108     }
109   }
110 }
```

Listing A.11: Implementierung CustomerPart POJO

A.8 Implementierung Service Modell und Service Interface

```
1  public class Car {
2    private int carID;
3    private String carName;
4
5    public Car(String name , int id)  {
6      carName=name;
7      carID=id;
8    }
9    public int getCarID() {
10     return this.carID;
11   }
12   public void setCarID(int carID) {
13     this.carID = carID;
14   }
15
16   public String getCarName() {
17     return carName;
18   }
19   public void setCarName(String name) {
20     this.carName = name;
21   }
22 }
```

Listing A.12: Implementierung Service Model Car

```
1  public class RentalOffice {
2    private int retalOfficeID;
3    private List<Car> cars = new ArrayList<Car>();
4
5    public int getRentalOfficeID() {
6      return this.rentalOfficeID;
7    }
8    public void setRentalOfficeID(int rentalOfficeID) {
9      this.rentalOfficeID = rentalOfficeID;
10   }
11
12   public void addCar(Car car) {
13     cars.add(car);
14   }
15
16   public List<Car> getCars() {
17     return cars;
18   }
19 }
```

Listing A.13: Implementierung Service Model Office

```
1  public interface ICarRentalModel {
2    List<RentalOffice> getRentalOffices();
3  }
```

Listing A.14: Implementierung Service Interface

A.9 Implementierung Renderer und RenderFactory

```
1  public class MyMapRenderer extends SWTPartRenderer {
2
3
4    public static final String ID = "de.preinhol.javascript.maps.view";
5    private static List list;
6
7    @Inject IShellProvider shellProvider;
8    @Inject Logger logger;
9
10   @Override
11   public Object createWidget(MUIElement element, Object parent) {
12
13   Bundle bundle=Platform.getBundle("de.preinhol.example.e4.maps.generate");
14   URL fileURL = bundle.getEntry("html/map.html");
15   File file = null;
16     try {
17       file = new File(FileLocator.resolve(fileURL).toURI());
18     } catch (URISyntaxException e1) {
19       e1.printStackTrace();
20     } catch (IOException e1) {
21       e1.printStackTrace(); }
22
23     final Composite mapComposite =new Composite((Composite)parent,SWT.NONE);
24     mapComposite.setLayout(new GridLayout(1,false));
25
26     final Browser browser = new Browser(mapComposite, SWT.NONE);
27
28     browser.addControlListener(new ControlListener() {
29       public void controlResized(ControlEvent e) {
30 //          Javascript zum setzen der Hoehe und Breite
31       browser.execute("document.getElementById('map_canvas').style.width="
32             + (browser.getSize().x - 20) + ";");
33       browser.execute("document.getElementById('map_canvas').style.height="
34             + (browser.getSize().y - 20) + ";");
35       }
36
37       public void controlMoved(ControlEvent e) {  }
38     });
39
40     browser.setUrl(file.toURI().toString());
41
42     GridData data = new GridData(SWT.FILL,SWT.FILL,true,true);
43     browser.setLayoutData(data);
44     mapComposite.setLayoutData(data);
45     return mapComposite;
46   }
47   /**
48    * Fokus an den Controller des Part weitergeben.
49    */
50   public void setFocus() { }
51 }
```

Listing A.15: Implementierung Renderer

```
1  public class MyRendererFactory extends WorkbenchRendererFactory {
2
3  private MyMapRenderer mapRenderer;
4  private ContributedPartRenderer partRenderer;
5
6  @Override
7  public AbstractPartRenderer getRenderer(MUIElement uiElement,Object parent)
8  {
9
10     if (uiElement instanceof OpenStreetMap) {
11       if (mapRenderer == null) {
12         mapRenderer = new MyMapRenderer();
13         super.initRenderer(mapRenderer);
14       }
15       return mapRenderer;
16     }
17     if(uiElement instanceof OSMContainer){
18       if(partRenderer == null){
19         partRenderer = new ContributedPartRenderer();
20         super.initRenderer(partRenderer);
21       }
22       return partRenderer;
23     }
24
25     return super.getRenderer(uiElement, parent);
26   }
27 }
```

Listing A.16: Implementierung RenderFactory

A.10 Screenshots Kontakt Manager alt/neu

Suche

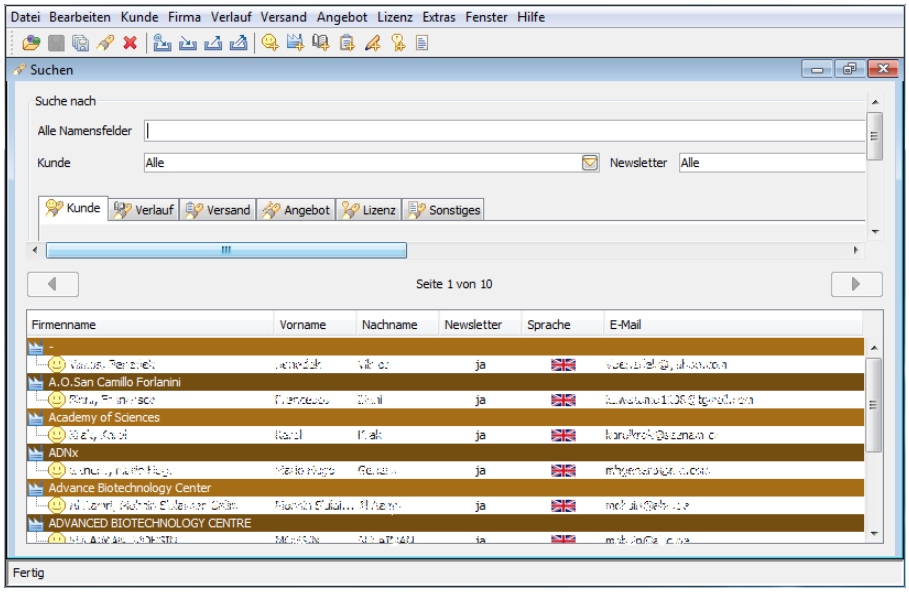

Abbildung A.3: Screenshot Suche Kontakt Manager

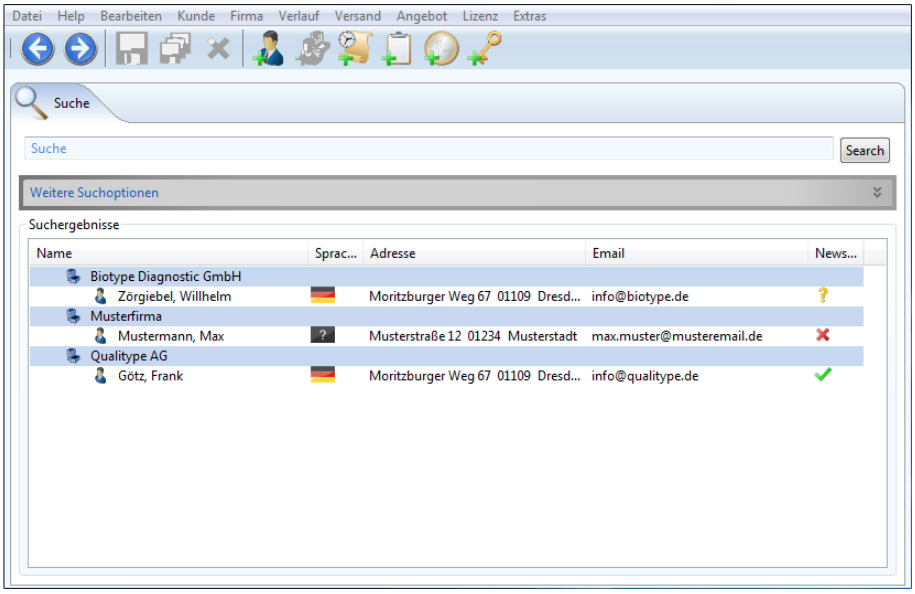

Abbildung A.4: Screenshot Suche Kontakt Manager 2

Kontaktübersicht

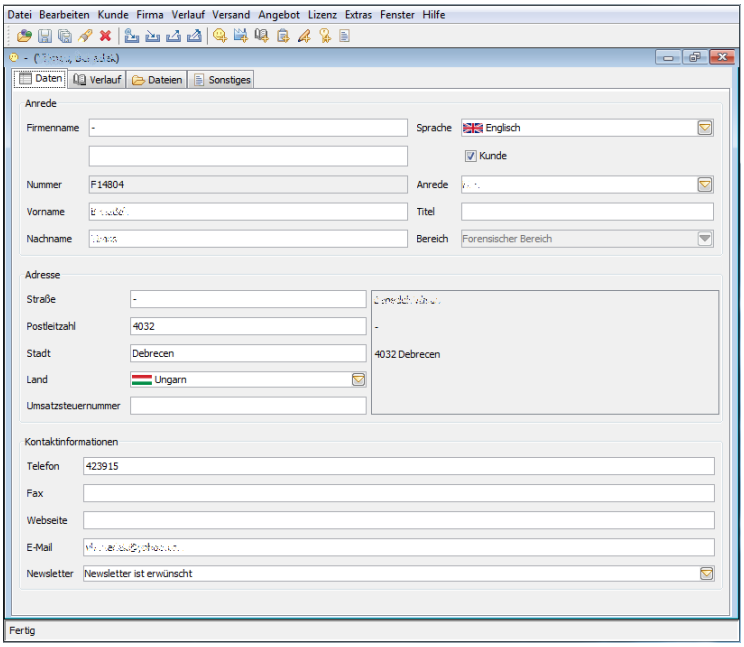

Abbildung A.5: Screenshot Kontaktansicht Kontakt Manager

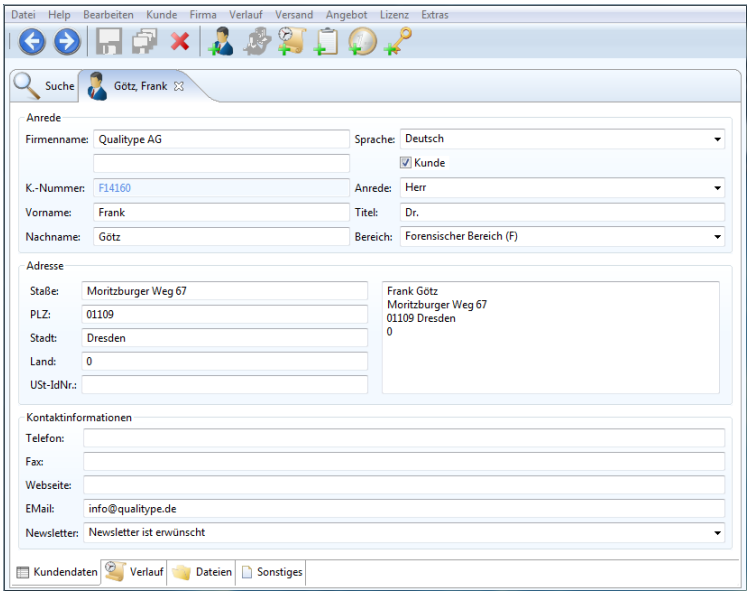

Abbildung A.6: Screenshot Kontaktansicht Kontakt Manager 2

Verlaufsübersicht

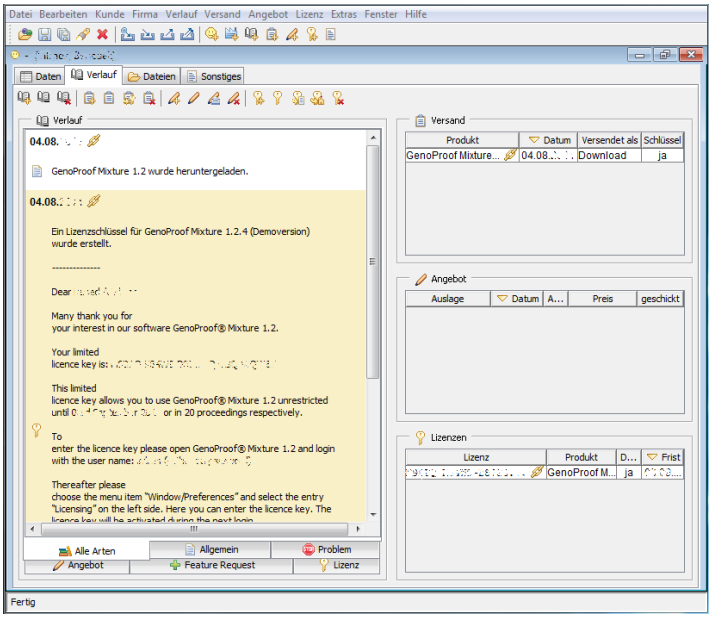

Abbildung A.7: Screenshot Verlaufsübersicht Kontakt Manager

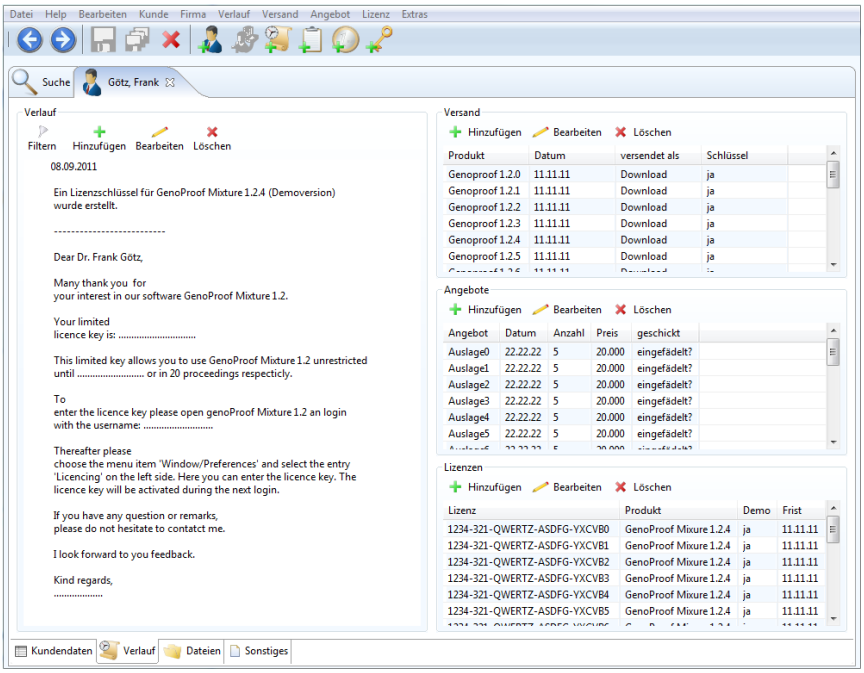

Abbildung A.8: Screenshot Verlaufsübersicht Kontakt Manager 2

B

Inhalt der CD

Auf der beigelegten CD befinden sich neben dem elektronischen Exemplar dieser Arbeit folgende Ordner:

- Grafiken
  - PNG - Grafiken in normaler Größe im PNG-Format
  - PNG BIG - Grafiken in großer Größe im PNG-Format
  - SVG - Grafiken im Vektorformat
- JBoss e4 Test
  - JBossTest - Eclipse Projekt der JBoss-Testapplikation
  - JBossTest.zip - gepacktes Projekt der JBoss-Testapplikation
- Neuer Kontaktmanager
  - ContactManager2.0 - Eclipse Projekt des neuen Kontaktmanagers
  - ContactManager2.0.zip - gepacktes Projekt des neuen Kontaktmanagers
  - e4_JBoss_Arch.svg - Grafik der Architektur im Vektorformat
  - Icons - Icons des neuen Kontaktmanagers im Vektorformat
- Quellen - Papers und Präsentationen zum Thema der Arbeit

C
Literaturverzeichnis

1 Einleitung

[Brock97] Brockhaus - Die Enzyklopädie: in 24 Bänden, Band 6, 20., überarbeitete und aktualisierte Auflage, Leipzig [u.a.], 1997, S. 716
http://www.evaluieren.de/evaluat.ion/definiti.htm
zuletzt geöffnet am 26.09.2011

[Umsta01] Evaluierungsdefinition
http://www.ib.hu-berlin.de/~wumsta/wistru/definitions/dk6.html
zuletzt geöffnet am 26.09.2011

2 Entwicklungsgrundlagen der Eclipse Rich Client Platform

[Riehle00] Dirk Riehle. Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509. Zürich, Switzerland, ETH Zürich, 2000.
http://dirkriehle.com/computer-science/research/dissertation/diss-a4.pdf
zuletzt geöffnet am 17.06.2011

[ShaHu06] Shan, Hua, Taxonomy of Java Web Application Frameworks
http://portal.acm.org/citation.cfm?id=1190953
zuletzt geöffnet am 17.06.2011

[GruTh00] Volker Gruhn, Andreas Thiel: Komponentenmodelle . DCOM, Javabeans, Enterprise Java Beans, CORBA Addison-Wesley, 2000

[Pree97] Wolfgang Pree: Komponentenbasierte Softwareentwicklung mit Frameworks. dpunkt.verlag, 1997

[WüHar08] Gerd Wütherich, Nils Hartmann, Bernd Kolb, Matthias Lübken Die OSGi Service Platform – Eine Einführung mit Eclipse Equinox dpunkt.verlag, 2008

[OSGi11] The OSGi Alliance, OSGi Service Plattform Core Specification, Release 4.3, April 2011
http://www.osgi.org/download/r4v43/r4.core.pdf
zuletzt geöffnet am: 11.07.2011

[Seeber08] Heiko Seeberger, OSGi-Serie ab Javamagazin, Dezember 2008, online unter:

http://it-republik.de/jaxenter/artikel/Erste-Schritte
-mit-OSGi-2077
http://it-republik.de/jaxenter/artikel/Erste-Schritte
-mit-OSGi--Teil-II-2078.html
http://it-republik.de/jaxenter/artikel/OSGi-in-kleinen-Dosen-%
96-Bundles-und-Life-Cycle-2118.html
http://it-republik.de/jaxenter/artikel/Services-%
E0-la-OSGi-2236.html
zuletzt geöffnet: 05.07.2011

[Seeber09] Heiko Seeberger, Quantensprung für Equinox, Javamagazin Juni 2009, online unter:
http://it-republik.de/jaxenter/artikel/
Quantensprung-fuer-Equinox-2381.html
zuletzt geöffnet am 12.07.2011

[Athor09] Arthorne, IBM Canada Inc., e4 technical Overview, Juli 2009
http://www.eclipse.org/e4/resources/e4-whitepaper.php
zuletzt geöffnet am 18.07.2011

3 Evaluierung des Eclipse RCP-Frameworks 4.1

[e4Evang] E4 Wiki Evangelism
http://wiki.eclipse.org/E4/Evangelism
zuletzt geöffnet am 17.09.2011

[Longm08] EMF: Eclipse Modeling Framework, Addison-Wesley Longman, Amsterdam; Auflage: 2nd Revised edition (REV), 16. Dezember 2008

[Knebe09] Knebel, Data Profiling mit Eclipse. Von den Grundlagen zum Prototypen, Hamburg, Diplomica Verlag GmbH, 2009

[BacGr05] Bacvanski, Graff, Mastering Eclipse Model Framework, Pesentation, EclipseCon 2005
http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf
zuletzt geöffnet am 13.05.2011

[Vogella10] Vogel Lars, Eclipse e4 Overview, Präsentation, 2010, Folie 12
http://www.slideshare.net/LarsVogel/eclipse-e4-overview
zuletzt geöffnet am: 27.09.2011

[SchiBo10] Schindel, Bokowski, Die Schaltzentrale von e4, Java Magazin, September 2010, online unter:
http://it-republik.de/jaxenter/artikel/Die-Schaltzentrale-von-e4-3321.html
zuletzt geöffnet am: 13.05.2011

[Fowler04] Fowler, Inversion of Control Containers and the Dependency Injection pattern, 23 January 2004
http://martinfowler.com/articles/injection.html
zuletzt geöffnet am 06.07.2011

[OatLu08] Oates, Langer, Wille, Lueckow, Bachlmayr, Spring & Hibernate: Eine praxisbezogene Einführung, Carl Hanser Verlag; Auflage: 2., aktualisierte Auflage, 3. April 2008

[e4DiWiki] E4 Wiki Dependecy Injection
http://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection
zuletzt geöffnet am 26.09.2011

[JSR330] Java Specification Request 330
http://jcp.org/aboutJava/communityprocess/final/jsr330/index.html
zuletzt geöffnet am 07.07.2011

[Straub11] Straube, inversion of control - dependency injection, blog, 14. Januar 2011
http://www.christian-straube.de/2011/01/11/inversion-of-control-dependency-injection/
zuletzt geöffnet am 06.07.2011

[Schin11a] Schindl, e4 – Fundamental Overview on the Eclipse 4 Application Platform, blog, 16. März 2011
http://tomsondev.bestsolution.at/2011/03/16/

e4-fundamental-overview-on-the-eclipse-4-application-platform/
zuletzt geöffnet am 20.07.2011

[Schin11b] Schindl, Eclipse 4.1 Application Platform - A plattform for anyone, Blog, 21. Juni 2011
http://tomsondev.bestsolution.at/2011/06/21/
eclipse-4-1-application-platform-a-platform-for-anyone/
zuletzt geöffnet am 20.07.2011

[Schin11c] Schindl, Enhanced RCP: How views can communicate – The e4 way
http://tomsondev.bestsolution.at/2011/02/07/
enhanced-rcp-how-views-can-communicate-the-e4-way/
zuletzt geöffnet am 30.09.2011

[CompLa] Eclipse Forum e4
http://www.eclipse.org/forums/index.php/m/682554/
zuletzt geöffnet am 15.06.2011

[BugCo] Bugbericht zum Kompatibilitätslayer
https://bugs.eclipse.org/bugs/show_bug.cgi?id=317912
zuletzt geöffnet am 15.06.2011

[E4Ctx] E4 Wiki IEclipse Context
http://wiki.eclipse.org/E4/Contexts
zuletzt geöffnet am 02.10.2011

[E4Add] E4 Wiki Addons Komponenten
http://wiki.eclipse.org/Eclipse4/RCP/Modeled_UI/Addons
zuletzt geöffnet am 03.10.2011

[E4EAS] E4 Wiki Eclipse Application Services
http://wiki.eclipse.org/Eclipse4/RCP/EAS
zuletzt geöffnet am 23.09.2011

[Schinl10] Tom Schindl,Präsentation, Democamp München 2010, 23.11.2010
http://wiki.eclipse.org/Eclipse_DemoCamps_November_2010/Munich
zuletzt geöffnet am 05.10.2011

[Schin11c] Schindl, e4 – Fundamental Overview on the Eclipse 4 Application Platform
http://tomsondev.bestsolution.at/2011/03/16/
e4-fundamental-overview-on-the-eclipse-4-application-platform/
zuletzt geöffnet am 02.09.2011

[Schind11d] Schindl, e4 – Use Eclipse 4.1 Application Platform but not SWT, Blog, März 2011

http://tomsondev.bestsolution.at/2011/03/02/
e4-use-eclipse-4-1-application-platform-but-not-swt/
zuletzt besucht am 04.08.2011

[Vogel10] Vogel, Eclipse e4 Renderer – Google Maps, November 2010
http://www.vogella.de/blog/2010/11/09/eclipse-e4-renderer/
zuletzt geöffnet am 02.08.2011

[Vogel11a] Vogel, e4 Tutorial, Contibution
http://www.vogella.de/articles/EclipseE4/ar01s15.html
zuletzt geöffnet am 04.10.2011

[Vogel09] Vogel, Developer Papercuts, Eclipse e4 – Styling your UI with css
http://www.vogella.de/blog/2009/08/18/eclipse-e4-css/
zuletzt geöffnet am 26.08.2011

[Toedt10a] Toedter, e4 Preview: User Interface Styling mit CSS, Artikel, März 2010
http://it-republik.de/jaxenter/artikel/e4-Preview-User-
Interface-Styling-mit-CSS-2924
zuletzt geöffnet am 25.08.2011

[Toedt10b] Toedter, CSS Theming for Eclipse RCP 3.x, Blog, August 2010
http://www.toedter.com/blog/?p=295
zuletzt geöffnet am 30.08.2011

[Sternb11] Sternberg, RAP in Indigo, Artikel, Juli 2011
http://.it-republik.de/jaxenter/artikel/RAP-in-Indigo
-Die-neue-Auflage- der-Rich-Ajax-Platform-3910.html
zuletzt geöffnet am 30.07.2011

[E4RAP] E4 Wiki, RAP Integration
http://wiki.eclipse.org/E4/RAP_Integration/Experimental
zuletzt geöffnet am 20.07.2011

[E4RapIn] http://wiki.eclipse.org/E4/RAP_Integration
zuletzt geöffnet 05.10.2011

4 Umstellung komplexer bestehender Anwendungen

[SchBlog] Tom Schindl, Blog, 2011
http://tomsondev.bestsolution.at/
zuletzt geöffnet am 12.10.2011

[VogBlog] Vogel Lars, Blog, 2011
http://www.vogella.de/blog/
zuletzt geöffnet am 12.10.2011

[ToeBlog] Toeder Kai, Blog, 2011
http://www.toedter.com/blog/
zuletzt geöffnet am 12.10.2011

[E4Wiki] E4 Wikipedia, Eclipse Foundation, 2011
http://wiki.eclipse.org/E4
zuletzt geöffnet am 12.10.2011

[E4Foru] E4 Forum, Eclipse.org, 2011
http://www.eclipse.org/forums/index.php?t=thread&frm_id=12
zuletzt geöffnet am 12.10.2011

[IBM06] Scott Delap, Understanding how Eclipse plug-ins work with OSGi, Online Article, IBM, 06 Juni 2006
http://www.ibm.com/developerworks/library/os-ecl-osgi/index.html
zuletzt geöffnet am 12.10.2011

[Kutsc11] Kutschera, Ant, Eclipse e4 - from Client to Server and back again with MVC, Online Article, Javalobby dzone, 05 August 2011
http://java.dzone.com/articles/eclipse-e4-client-server-and
zuletzt geöffnet am 20.08.2011

[Vogel10b] Vogel Lars, Discouraged Access Warning, Google Group, November 2010
http://groups.google.com/group/vogella/browse_thread/thread/0691065b3129f884
zuletzt geöffnet am 11.08.2011

[WikiEcl] Wikipedia.de, Artikel History Eclispe(IDE), 2011
http://de.wikipedia.org/w/index.php?title=Eclipse_(IDE)&action=history
zuletzt geöffnet am 12.10.2011

[E4Bugs] Know Issues Eclipse 4.1, 2011
http://wiki.eclipse.org/Eclipse4.1/KnownIssues
zuletzt geöffnet am 25.09.2011

[Rubin10] Andrew Lee Rubinger, Bill Bork, Enterprise JavaBeans 3.1, Sixth Edition, September 2011, O'Reilly

[ClRu09] Eric Cleyberg, Dan Rubel, Eclipse Plug-ins, third Edition, 2009, Pearson Education Inc

D
Glossar

Annotation	Erweiterung des Quellcode mit auswertbaren Anmerkungen. Im Java Umfeld beginnen diese mit einem at-Zeichen., 21
Boilerplate Code	Häufig triviale Codefragmente, die an vielen Stellen in mehr oder weniger unveränderter Form benötigt werden., 53
Bootstrap	Durch einen einzelnen Prozess, wird ein zunehmend komplexeres System erzeugt. Das System startet sich in gewisser Weise selbst., 6
Bundle	Bezeichnet die Komponenten im OSGi Umfeld., 6
Commands	Abstraktes Kommandos, wie ein Speichern-Kommando, der Speichervorgang an sich ist implementationsabhängig, 13
CRUD	Create, Read, Update, Delete - die grundlegenden Operationen von Datenzugriffen, 59
CSS	Cascading Style Sheets sind quasi eine deklarative Sprache für Stilvorlagen., 43
Dependecy Injection	Entwurfsmuster nach Fowler, dient in einem objektorientierten System dazu, die Abhängigkeiten zwischen Komponenten oder Objekten zu minimieren., 18
Dependency	Abhänigkeit zwischen zwei Komponenten., 8
DI	Abkürzung, siehe Dependency Injection, 18

e4	Technologie Brutstätte, bezeichnet den Ort an dem neue Technologien für Eclipse 4.x entwickelt werden. Ziel ist es mit auf Grundlage von e4 eine RCP 2.0 Plattform zu schaffen., 13
E4AP	siehe Eclipse 4 Application Platform, 27
EAP	siehe Eclipse 4 Application Platform, 27
Eclipse 4 Application Platform	Die Eclipse 4 Application Platform ist die Grundlage jeder e4-Anwendung., 27
EJB	Enterprise JavaBeans- Komponenten innerhalb eines Java Enterprise-Servers., 59
EMF	Das Eclipse Modeling Framework pragmatischer Ansatz von modellgetriebener Software Entwicklung mit dem Fokus auf Codegenerierung und Importfunktionalitäten., 13
Engine	Eng verbunden mit einem anderen Prozess, z.B. dem Rendering, die Engine führt dabei im Hintergrund die notwendigen und oft komplexen Berechnungen aus., 45
Entity	Als Entität (auch Informationsobjekt) wird in der Datenmodellierung ein eindeutig zu bestimmendes Objekt bezeichnet, über das Informationen gespeichert oder verarbeitet werden sollen., 59
Entwurfsmuster	Entwurfsmuster sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in Softwarearchitektur und Softwareentwicklung., 18
Equinox	Equinox ist ein, nach der OSGi-Kernspezifikation implementiertes, Java-basiertes Framework. Es bildet das Gerüst der integrierten Entwicklungsumgebung (IDE) Eclipse., 6
Extension Point	Erweiterungen welche die Möglichkeiten bieten Funktionalitäten eines Plug-ins anderen zur Verfügung zu stellen., 8
Framework	Programmierstruktur, welche die gesammelte Erfahrung, wie Architektur und Implementierung der meisten Applikationen einer Domäne aussehen sollten, repräsentiert., 5
Gradient	Begriff aus der Grafik, der einen Farbverlauf entlang einer Linie bezeichnet., 44

GUI	Graphical User Interface - die grafische Benutzeroberfläche, 8
IDE	Eine integrierte Entwicklungsumgebung ist eine Sammlung von Anwendungsprogrammen, mit denen die Aufgaben der Softwareentwicklung bearbeit werden können., 8
Inversion of Control	Umgekehrung der Steuerung, Paradigma und Definitionskriterium von Frameworks, 18
Komponente	Die Komponente im Softwareumfeld, ist ein wohldefinierte Teil einer Software mit hohem inneren Zusammenhang und loser Kopplung zu anderen Komponenten., 6
Launcher	Ein Programm zum Starten von Programmen., 6
Muster	siehe Entwurfsmuster, 18
OSGi	OSGi stand ursprünglich als Abkürzung für Open Service Gateway Initiative, heute steht der Begriff für sich bzw. als Teil von OSGi Service Platform, Titel der aktuellen Spezifikation (Version 4.1)., 6
OSM	Open Street Map – ist ein freies Projekt, das für jeden frei nutzbare Geodaten sammelt (Open Data)., 47
Part	Innerhalb von e4 neu eingeführte Zusammenfassung von Editoren und Views. Parts sind abstrakt. Erhalten ihren Inhalt durch POJOs., 15
Plug-in	Module oder Komponenten im RCP Umfeld., 8
POJOs	Plain Old Java Object – ganz normales Java Objekt ohne Abhäningkeiten., 15
Property	Eine Property bezeichnet eine Eigenschaft. Meist besitzt diese einen Namen und einen Wert., 26
RAP	Rich Ajax Platform von Eclipse, auf Java basierendes Plugin zur Entwicklung von Web 2.0 Anwendungen., 51

RCP	Rich Client Plattform– eine anpassungsfähige und erweiterbare Entwicklungsumgebung zur Entwicklung von Rich-Client-Applikationen., 4
Renderer	Ein Renderer ermöglicht das Anzeigen von Inhalten auf einer GUI auf Grundlage eines Modells (z.B. dem Applikationsmodell), 45
SDK	Software Development Kit – Sammlung von Werkzeugen und Anwendungen, um eine Software zu erstellen, meist inklusive Dokumentation, 13
Softwarekomponente	siehe Komponente, 6
SWT	Das Standard Widget Toolkit –eine Bibliothek für die Erstellung grafischer Oberflächen mit Java., 8
UI	User Interface – Benutzerschnittstelle, nicht zwangsweise grafisch., 14
URI	Uniform Resource Identifier – Ein eindeutiger Bezeichner für eine Ressource., 15
View	Eine Ansicht typisch für das 3.x Framework. Als Ansichten können dabei Listen, Tabellen, Baumstrukturen etc. gemeint sein. Views erlauben keine Eingabe von Daten., 8
Widget	Widget (Window gadget) bezeichnet eine Komponente eines grafischen Fenstersytems., 13
Workbench	Die Workbench beschreibt ein abstraktes Arbeitsumfeld, vorstellbar als leerer Applikationsrahmen., 13
XMI	XML Metadata Interchange ist ein Standard der zunehmend als Austauschformat zwischen Software-Entwicklungswerkzeugen verwendet wird., 15
XML	Die Extensible Markup Language ist eine Sprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten., 15
Zugriffsmodifikator	In Java sind das private, protected und public., 21

E

Erklärung zur selbständigen Anfertigung

Erklärung:

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Bearbeitungsort, Datum Unterschrift